AFRL-VA-WP-TM-2005-3083

TEMPORALLY AWARE REACTIVE SYSTEMS

Dr. Mark P. Jones

Oregon Graduate Institute (OHSU)
School of Science and Engineering
20000 NW Walker Road
Beaverton, OR 97006-8921

MARCH 2005

Final Report for 01 July 2000 – 01 March 2004

Approved for public release; distribution is unlimited.

STINFO FINAL REPORT

AIR VEHICLES DIRECTORATE
AIR FORCE MATERIEL COMMAND
AIR FORCE RESEARCH LABORATORY
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7542

NOTICE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Air Force Research Laboratory Wright Site Public Affairs Office (AFRL/WS) and is releasable to the National Technical Information Service (NTIS). It will be available to the general public, including foreign nationals.

PAO Case Number: AFRL/WS 05-1124, 09 May 2005.

THIS TECHNICAL REPORT IS APPROVED FOR PUBLICATION.

/s/

Raymond A. Bortner
Senior Electronic Engineer

/s/

Michael P. Camden, Chief
Control Systems Development and
Applications Branch

/s/

BRIAN W. VAN VLIET
Chief, Control Sciences Division
Air Vehicles Directorate

This report is published in the interest of scientific and technical information exchange and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
March 2005	Final	07/01/2000– 03/01/2004

4. TITLE AND SUBTITLE
TEMPORALLY AWARE REACTIVE SYSTEMS

5a. CONTRACT NUMBER
F33615-00-C-3042

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
69199F

6. AUTHOR(S)
Dr. Mark P. Jones

5d. PROJECT NUMBER
A04I

5e. TASK NUMBER

5f. WORK UNIT NUMBER
0C

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Oregon Graduate Institute (OHSU)
School of Science and Engineering
20000 NW Walker Road
Beaverton, OR 97006-8921

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Vehicles Directorate
Air Force Research Laboratory
Air Force Materiel Command
Wright-Patterson Air Force Base, OH 45433-7542

Defense Advanced Research Projects Agency (DARPA/IXO)
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/VACC

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-VA-WP-TM-2005-3083

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

14. ABSTRACT

Real-time embedded software development is recognized as a significant cost and schedule driver for many of today's advanced–and highly software-centric–military systems. This project targets these issues directly by developing techniques to enable rapid construction and greater reuse in embedded systems while also delivering reliability and performance improvements. In particular, these benefits are a result of (a) the ability to analyze the behavior and correctness of real-time software; (b) the ability to construct individual software components or to develop large configurations of multiple components more rapidly by leveraging high-level, domain-specific abstractions (a productivity benefit); (c) the ability to enforce predictable and graceful degradation behavior during overload (a reliability benefit); and (d) the ability to operate closer to resource saturation (a performance benefit due to support for graceful degradation).

15. SUBJECT TERMS
Real-time software; Static Scheduling; Domain-specific languages; Static Checking; Overload tolerance; Graceful degradation

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Raymond A. Bortner
Unclassified	Unclassified	Unclassified	SAR	246	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-8292

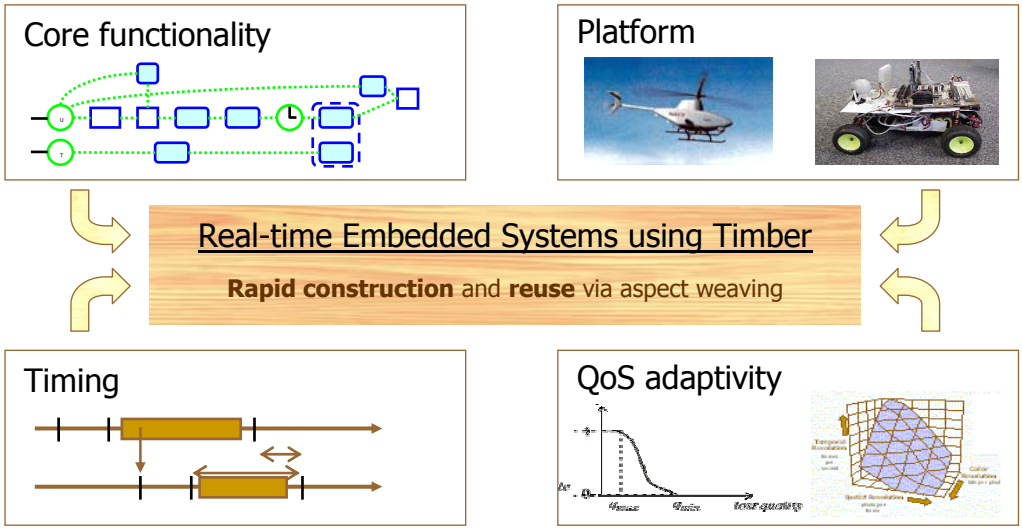
Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

1 Overview and Approach

The original goals of the "Temporally Aware Reactive Programs" project (referred to informally as "Project Timber") were to develop new programming language technology and analysis tools to support a *compositional* approach to programming of *large-scale*, *real-time*, *embedded* systems with high assurance of non-functional aspects of behavior. Our work was performed in the context of the DARPA PCES program ("Program Composition for Embedded Systems")

The project focused, in particular, on the central role of *time* in embedded systems design. This includes aspects of behavior that require hard, real-time guarantees as well as those that are driven by softer, quality of service (QoS) requirements with policies for overload tolerance to support adaptivity and graceful degradation in resource constrained, real-rate systems. Our approach centered on the development of a high-level, reactive programming language for component implementation and system integration in which temporal behavior is declared explicitly. We designed this language, called *Timber* (Time as a basis for embedded real-time systems), to allow programmers to separate and describe different aspects of system behavior at a high-level, relying on analysis and compiler technology to weave these specifications together to build working



systems. The diagram above illustrates this concept, suggesting how aspects of core functionality (described by program code), timing (captured by temporal constraints), QoS (specified in terms of utility functions and statistical distributions), and platform specifics might be combined to support rapid construction of real-time, embedded systems with guaranteed temporal behavior.

Conventional middleware provides high-level, portable services and mechanisms that can be used across a range of applications. In a similar way, Timber provides a kind of "language middleware" that leverages the advantages of domain-specific languages in providing safe, expressive, and efficient encapsulation of time-oriented and QoS programming patterns. For example, Timber programs are written in terms of *events* and *reactions*—to which appropriate timing constraints can be assigned—and benefit from an implicit treatment of threading with no blocking operations, which avoids the problems of priority inversion. But Timber is also very expressive; for example, communication with callbacks is facilitated by the use of first-class reactions and a monadic type system. In the design of Timber, scheduling and concurrency play key roles in weaving together aspects of functional and extra-functional behavior. For hard real-time reactions we developed techniques for calculating static schedules from explicit temporal constraints, and for applying these to obtain firm static guarantees of timeliness for both periodic and aperiodic events. For soft real-time reactions we developed techniques for applying dynamic and adaptive scheduling, driven by utility functions to provide optimal scheduling with guarantees of statistical QoS properties.

In the early stages of the project, we developed a mathematically rigorous model of computation for timed reactive systems to serve as a basis for the design of the Timber language and as a foundation for static scheduling of time sensitive reactions. At the same time, as a driver for investigation of techniques for adaptation and graceful degradation, we worked on the development of a general priority-progress streaming (PPS) model and its application to a video-streaming pipeline that used software feedback to adapt and respond in real-time to changing network bandwidth and varying CPU overhead in accordance with dynamically specified user QoS preferences.

The introduction of two Open Experimental Platforms in the PCES program provided opportunities to focus our research efforts and to demonstrate their applicability in a context that was common to all of the projects in the program. In particular, we worked to integrate our video pipeline software in the context of the BBN UAV OEP, adding support for live video streaming and demonstrating its potential for flexible and adaptive flow control in assuring end-to-end QoS in UAV video reconnaissance. We also demonstrated that we could use Timbot, an autonomous robot vehicle

equipped with a video camera, as an additional UAV simulation host that could reliably execute demanding, time-critical control software (written in Timber) on the same platform as the rate-critical components of the live video streaming software.

As the program continued to mature, we began to develop generalizations of the PPS approach to adaptive streaming that was used in our video pipeline (i) to support additional dimensions of adaptation (such as region of interest, color space, speed and precision of robot navigation, power consumption, etc…); (ii) to accommodate different kinds of data (e.g., audio, still images, streaming GIS/terrain data, etc…); and (iii) to offer increasing levels of scalability. At the same time, we applied Timber to the construction of a new domain-specific language (DSL) for component configuration in the context of the Boeing OEP. Although this did not require the facilities of Timber for specifying time-critical behavior, our work with the resulting DSL showed significant benefits from the use of a domain-specific language, including (i) a significant reduction in code side (DSL programs are typically between a tenth to a thirtieth of the size of their original descriptions in the Boeing OEP), enabling better scalability and increased productivity; (ii) support for modularity and reusability, facilitating the description and construction of large configurations by the members of a team; (iii) substantial improvements in reliability resulting from the ability to detect and eliminate many errors through automatic static checking and analysis; and (iv) increased flexibility, allowing configuration data to be more easily exported to other tools for visualization, debugging, and other purposes. One of the last pieces of work before the project ended was to show how the functionality of the DSL could be packaged in a standalone library, providing a mechanism for dynamic reconfiguration, which is a significant new capability with applications to increasing mission agility, enhancing fault tolerance, facilitating sophisticated load balancing, and enabling fine-grained customization.

2 Project Chronology

Real-time embedded software development is recognized as a significant cost and schedule driver for many current, advanced, and highly software-centric military systems. Although some promising programming technologies have been developed—including real-time open systems and middleware, aspect-oriented programming, and patterns—their potential has not yet been realized due to limited technology support and developing code for such applications is still a time-intensive and highly error prone process.

The DARPA PCES program (Program Composition for Embedded Systems) was developed to address these problems, seeking ground-breaking technologies for new forms of program composition to facilitate rapid and reliable construction of large-scale, flexible software systems. Several high-level goals were identified, including (i) the development of techniques for implementing and assuring key properties/aspects of real-time embedded systems such as concurrency, synchronization, timing, and fault tolerance; and (ii) the development of "aspect weaving" tools and technologies that would allow the construction of sophisticated, embedded systems software by combining code from reusable "aspect libraries" with the code that implements only the desired algorithmic functionality of a specific target system. The original focus of the PCES program was on program analysis and (property-preserving) composition techniques that could be used cooperatively to support real-time programming. This emphasis was reflected in the identification of four technical topic areas: aspect suites for embedded systems; program representation and analysis; program transformation systems; and stage management systems.

Within the PCES program, OGI was awarded a contract to develop tools and techniques to support a comprehensive, new approach to building and analyzing software for embedded systems, with a particularly strong emphasis on temporal behavior and constraints. This approach was based upon a model of computation called Timed Reactive Systems (TRS) in which behavior is specified as a collection of time-sensitive reactions to events that occur in the operational environment. When software is specified in this model, dynamic reactions can be statically scheduled, based upon an automated analysis of the program, to implement real-time reactions that satisfy the declared timing

constraints. The calculation of a schedule is a form of aspect weaving that automatically interleaves different aspects of system behavior together in a working system. Moreover, successful computation of a schedule provides assurance that hard timing constraints on system behavior can (and will) be met. At the same time, failure to compute a schedule provides critical, design-time feedback to system developers by identifying infeasible designs, and, in many cases, highlighting specific issues of a software and/or hardware architecture that must be addressed in order to obtain the required behavioral guarantees.

This approach is particularly well-suited to real-time, embedded systems development, and it is also rather different to traditional approaches that have tended to assume that all time-critical processes are periodic, or else that they can be described in terms of prioritized events rather than more direct temporal constraints. In addition to developing the foundations of TRS, our work also called for the design and implementation of a programming language called Timber (**TIM**e as a **B**asis for **E**mbedded **R**eal-time systems) to support programming in this model. A new language design was needed because there were no available languages with the necessary features to support a compositional approach to programming real-time embedded systems in line with the TRS model. From the beginning, however, we intended to leverage key features of existing languages that we considered to be necessary and appropriate for high-assurance and embedded software development, including: a powerful, functional language core; an object-oriented, imperative framework of reactive computations; and a strong, monadic type system with the ability to track and delimit the scope of computational effects automatically within component types. These same facilities allow us to treat Timber as a *domain-specific language* for high-assurance, adaptive, and portable real-time embedded systems, supporting encapsulation of time-oriented and quality of service programming patterns with static analysis to support behavioral guarantees. The Timber language played a central role in our work and this was reflected by the fact that we often refer to the overall project as "Project Timber" instead of using the formal title of "Temporally Aware Reactive Programs."

To gain empirical feedback on the technology and concepts that we developed, our work included plans to use them in the construction of embedded systems for two different application domains: active network routers and real-time robot control. A key concern in the active networking

community is the question of how to support extension and customization of packet filters without compromising on security or throughput. Examples of customization include application-specific rate shaping policies, routing, data synchronization, and logging. For real-time robot control, we planned to focus on the task of controlling a mobile robot that would carry various sensors, including a video camera, and on streaming back live video over a wireless network. Because of resource limitations on such platforms (resulting, for example, from the sharing of a single CPU for hard real-time control as well as softer real-time streaming), and also because of changes in wireless network bandwidth resulting from shadowing or path loss as the robot moves, it is essential that such systems support adaptive QoS and resource management. This provides for graceful degradation of service, instead of system failure, in accordance with user-specified, and, potentially, varying quality preferences or requirements. In this area, we planned to build on work that had previously been conducted in the Quasar project at OGI as part of the Quorum program, which had developed techniques for software feedback and end-to-end QoS support. This also included the development of prototypes of multimedia pipelines that scale data flows in response to resource paucity or profusion while respecting end-user specified utility metrics, and to low-level feedback controllers that allocate resources such as CPU cycles or network bandwidth by monitoring application level progress.

The specific focus of the research activity, determining which ideas were to be pursued vigorously and which were not, was in flux throughout the duration of the project by agreement of the Government and OGI scientists so as to provide DARPA with the flexibility to achieve maximum benefit from the project towards its overall program objectives

A brief chronology of the project is provided in the following sections.

2.1 Project Year 1 (July 2000-June 2001)

The OGI contract for Project Timber began on July 20, 2000, and the OGI team explored worked in several different areas during this time, as described in the following subsections.

2.1.1 Investigation of Relevant Applications and Domains

One of the first requirements of the project was to identify, and more fully understand, and explore relevant applications and domains that would play a critical role in motivating, informing, and guiding the design of the Timber language. In particular, we focused on (i) the use of Infopipe and Quasar abstractions in the construction of real-rate adaptive video streaming; and (ii) the construction of embedded network routers, both on custom hardware developed by Intel, and on stock hardware based on the Click modular router that was developed at MIT. Towards the end of the first year, using the prototype interpreter that we had developed by that time, we were able to model the basic SEC flight control components in Timber. Examples like these continued to play an important part throughout the project.

We also began investigations in the area of dynamic adaptation and graceful degradation, working towards the long term goal of capturing these ideas in the form of a reusable "aspect library" or corresponding design patterns. We focused, in particular, on practical mechanisms for dynamic load estimation and overload tolerance including: time stamping and prioritization of data in packet streams; feedback control and dynamic scheduling; buffer level monitoring; and the role of utility functions in mapping between objective and subjective notions of quality. These techniques drew from, and (fed back) into ongoing development of the Quasar adaptive video pipeline software, and provided insights into key aspects of the design space for Timber in these domains.

2.1.2 Theoretical Foundations

To provide firm theoretical foundations for Timber, we developed the formal computational model of Timed Reactive Systems (TRS). This work was driven by the goal of using static analysis to enable strong and meaningful guarantees about the ability of a system to satisfy explicit timing constraints. We focused on reactive languages such as Esterel, Lustre and O'Haskell as significant influences for the design of Timber, and on the kinds of extension that would be needed to support the declaration of temporal constraints. In addition to formal modeling, the OGI team developed a prototype implementation of a static analysis and scheduling algorithm that takes, as input, the description of a reactive program, focusing on essential timing properties such as reaction times, inter-event latencies, and response times. The output provides a static schedule for the reactive

program, which describes how the program can be implemented without the overhead or uncertainty of dynamic scheduling. Most importantly, the existence of a static schedule tells us that the program can be implemented in such a way as to guarantee that its timing properties will be met. As such, the algorithm can be used, not only for program compilation, but also as a design-level tool to assess the feasibility of a given reactive program: the algorithm will not produce a static schedule if conformance to timing requirements cannot be guaranteed. In addition, by separating the description of timing behavior from other aspects of a reactive program's functionality, we enhance portability and reuse. For example, we can vary the specification of reaction times in different runs of the static scheduler to evaluate the feasibility of executing a program on different platforms, and to generate a different static schedule for each one.

2.1.3 Practical Implementation

Based on our investigations of relevant domains, theoretical foundations, and existing programming languages, we selected O'Haskell as a starting point for the design of Timber. We concluded that O'Haskell was the most appropriate choice because it already provided most of the features that we had identified as necessary components for our work, including: a high-level model of reactive programming and event-oriented communication; support for stateful concurrent objects; high-level programming abstractions, inherited from the functional programming language Haskell; and advanced but flexible type system features that guarantee safe execution of programs while also providing a basis for program analysis. The requirements for Timber, however, also necessitated several significant changes to O'Haskell. For example, to obtain predictable timing behavior, we concluded that an eager evaluation strategy would be needed in place of the non-strict, or lazy, evaluation strategy that is used in Haskell and O'Haskell. Most significantly, of course, the design of Timber also called for new mechanisms, not provided in O'Haskell, to allow aspects of programming behavior, particularly those related to time, to be expressed explicitly using language constructs rather than indirectly using API calls, or captured only implicitly in the way that a program is written. To maximize our ability to use O'Haskell, we recruited its original designer and implementor, Dr. Johan Nordlander, as a postdoctoral researcher. We also used his O'Hugs implementation of O'Haskell (which had, in turn, been built from the Hugs interpreter for

Haskell that had been developed by Dr. Mark Jones, the Project Timber PI) as the starting point for building a prototype Timber interpreter called `Ti`.

The Timber interpreter, `Ti`, allowed some forms of timing behavior to be captured by annotating the component reactions of a reactive program with execution deadlines. These annotations were used to control a modified dynamic scheduler embedded in the interpreter's runtime system, using an earliest deadline first (EDF) policy to ensure that timing constraints are met whenever possible. We coded several examples using these annotations to demonstrate their utility, and also to understand their limitations, for example, in expressing drift-free, periodic tasks. This work suggested further refinements and improvements to our original language design. For example, in collaboration with OGI scientists involved in the SEC (Software Enabled Control) program, we investigated and implemented an alternative notion of timing annotations that used a common baseline for propagated methods calls; we implemented tagging of events (messages) in `Ti` as an attempt to achieve better to control over yet undelivered messages (e.g., timeouts); and we worked on the problem of garbage collecting processes that can still be reached but whose output cannot be observed. These experiments provided further useful feedback to shape the design of Timber while also supporting efforts to determine how to the results of our work on static scheduling analysis could be incorporated into the dynamic EDF scheduler of our prototype.

Towards the end of the first year, we began working towards the development of a prototype compiler for the Timber language. At that time there was significant interest within the PCES program in the evolving (but, at that point, unimplemented) Real-Time Specification for Java (RTSJ). As a result, we focused on the potential for translating Timber into Java, especially with its particular mechanisms for supporting concurrent objects and asynchronous communication. We also investigated techniques for compiling the high-level constructs and timing annotations of Timber to make use of the facilities of the real-time Java APIs.

2.1.4 Static Analysis using Type Systems

As a complement to the use of static analysis for the purposes of computing static schedules, Timber was also conceived as a language with a rich static type structure that would allow flexible

definition, encapsulation, and application of user-defined datatypes and operations on those types. During the initial stages of development, we worked on the design and construction of modular type systems and type checkers. We expected to exploit this work in developing an expressive type system for Timber, and in deriving supporting type inference and analysis tools. We also started to investigate the role that subtyping can play in documenting and enforcing timing constraints between the components of a real-time, embedded software system. This included developing techniques that used subtyping and monadic types to capture communication protocols between components in infopipes and in information flow networks, such as the Click modular router.

2.2 Project Year 2 (July 2001-June 2002)

One of the problems that arose in a number of different projects within the PCES program during the first year was the lack of standard, program-wide examples that could provide: a platform for experimental testing; a basis for comparison and integration between projects; and a direct connection to potential adopters in relevant DoD domains. These issues were addressed by the program manager in the second year of the program with the introduction of two OEPs (Open Experimental Platforms):

- The BBN UAV OEP modeled aspects of a system that could receive video and image data from one or more remote (simulated) UAV sources and distribute it to one or more viewing stations, ATR units, etc. The BBN UAV OEP included an adaptive video streaming system executing on a CORBA-based infrastructure for quality of service adaptation, and was constructed using primarily C++ code in combination with custom contract and aspect description languages.
- The Boeing Avionics OEP, based on the Boeing Bold Stroke Architecture, focused on the challenges of configuring and analyzing large-scale component-based software for use in distributed real-time embedded systems. The Boeing OEP included a CORBA-based infrastructure for event-based computation and was implemented using a combination of technologies including C++, UML, and XML.

Both of these OEPs were based on CORBA technology using the TAO real-time object broker and the Adaptive Communication Environment (ACE).

The introduction of the OEPs provided a focus for all of the researchers in the PCES program, and also signaled a change in emphasis towards integration, empirical evaluation, and technology transfer of existing technologies and away from the development of new technologies. These changes had significant impact on our technical focus in the second (and subsequent) years of the project, and led to us focusing our efforts in two areas. The first of these was in preparing our core technologies so that they could be used effectively as a basis for experimentation and as a guide to further development. The second was in evaluating and exploring the two OEPs to ensure a common context for us within the program and to identify opportunities where our specific technologies could be integrated or leveraged in collaborative efforts. We describe this work in more detail in the following subsections.

2.2.1 Language Definition

With the basic foundations and design in place, we worked to produce a draft language definition for Timber that was targeted to potential collaborators within the PCES program and, more broadly, was intended to serve as an introduction to and motivation for the key features of the language.

We also worked to establish a formal semantics for Timber to be used as a means of describing the language to others, to provide a foundation for our scheduling and analysis techniques, and to serve as a standard against which the correctness of our implementations could be assessed. These efforts resulted in a high-level semantics for real-time embedded systems based on a notion of time-sensitive reactive processes in which the actions of concurrent objects are executed subject to real-time constraints. Given accurate estimates of worst-case execution times (WCET), a system with this semantics can either be assured to satisfy a declared set of time constraints in all execution trajectories, or a possible counter-example to the constraint set can be shown. The semantics was developed using a process-algebra formalism so as to capture concurrency aspects of the execution framework, while abstracting away from low-level resource usage aspects (such as registers, data memory, instruction store, and IO buffers) that are often manifested in an abstract machine formalism.

2.2.2 Language Implementation

We continued to refine our Timber interpreter, `Ti`, and we used it to prototype several example applications, including the contract language in the BBN UAV OEP, and a simplified model of the Boeing OEP configuration language. The bulk of our efforts in this area, however, focused on the development of a new compiler for Timber producing executables for real-time platforms. (As an interpreter, Ti uses only a simulated notion of time, and cannot guarantee real-time performance.)

We had previously invested some effort in studying the RTSJ (Real-time Specification for Java) as a potential target for a Timber compiler, using its real-time features to provide the functionality needed for Timber executables. More specifically, we had hoped to use the open source reference implementation of RTSJ that was being developed by TimeSys to run on top of their Linux/RT operating system. Unfortunately, to avoid delay to our project, we were forced to abandon this approach because there was still no workable implementation of RTSJ available to us at that time. In particular, we had anticipated the need to add features to an RTSJ implementation, such as additional schedulers that are not guaranteed by the basic specification, so access to an open source version of RTSJ was crucial.

We therefore moved to a different implementation strategy, targeting the standard and widely implemented POSIX threads API so that our generated code could be executed on genuine real-time platforms—such as RT/Linux—as well as on platforms like standard Linux and MacOS X that support the basic APIs without guarantees of real-time behavior. In view of the goals of our project, we considered RT/Linux as the primary target of our compiler. Nevertheless, the ability to run compiled Timber programs on standard desktop operating systems is also very convenient during program development and functional testing.

The initial working prototype of our Timber compiler was completed in the second quarter of this project year. The compiler was written in a subset of Timber that is also compatible with Haskell so that we could use existing Haskell compilers for bootstrapping. The output of the compiler was ANSI C, augmented with calls to POSIX APIs and a small-runtime library that provides support for memory allocation, concurrency, event handling, and dynamic scheduling of reactions.

2.2.3 Static Analysis using Type Systems

Motivated again by program-wide interests in the RTSJ, we also began some work in the second year of the project to explore the potential for applying the techniques of region analysis and monadic type systems in the context of memory management for real-time Java. This resulted in a calculus for modeling scoped memory in the style of the RTSJ (Real-time Specification for Java) with a type-based analysis that allows all run-time checks on assignment (protecting against the possibility of dangling pointer errors) to be omitted, relying instead on static, compile-time tests to ensure safety. We intended to continue this work to evaluate its effectiveness in practice; to develop a more direct understanding of its expressivity and of its relationship to the RTSJ standard; and to add a prototype implementation of the analysis and its run-time support to one of our implementations of Timber. We suspended work in this area when the researcher focused on this task transferred to another institution, and decided not to resume this activity in light of the problems with availability of RTSJ implementations and the resulting de-emphasis of RTSJ in the PCES program.

2.2.4 OEP Integration

As a result of studying the BBN OEP, we discovered strong similarities between some aspects of Timber and the Contract Description Language that was used in BBN's QuO system, and we pursued several opportunities to build on this. For example, we investigated how our techniques for scheduling analysis could potentially be applied to analyze and guarantee feasibility of contracts. In addition, we developed a Timber version of the contract used in the BBN OEP demo, to demonstrate how Timber could be used to support more effective composition and reuse in contract descriptions. For example, using Timber, the recurring idiom in which a contract alternates between test and duty cycles to probe for resource availability can be abstracted out as a higher-order function or "pattern" in a library of reusable code.

We also investigated a general model of priority-progress streaming (PPS) for use in real-rate applications that provides practical mechanisms for dynamic load estimation and overload tolerance using time stamping and prioritization of data in packet streams. Such an approach allows for more flexible adaptation in applications such as video pipelines, where the annotations

on streamed data can be used to implement intelligent, selective packet dropping according to user-specified quality requirements.

These efforts were focused by two plans for integration and experimentation in the context of the BBN OEP demo. The first was to recode the original, strategy for controlling temporal resolution of MPEG video via frame dropping in the BBN OEP in terms of the more general model of PPS. The second was to modify the OEP to use a variant of the MPEG standard called SPEG that hasdbeen developed at OGI to allow dynamic adaptation in additional quality dimensions such as spatial and color resolution. The integration effort benefited significantly from a visit to the BBN site by OGI personnel, and the goals of both plans were realized. Successful integration of the OGI PPS software with the BBN pipeline was demonstrated at the PI meeting that was held in July 2002, and we were able to demonstrate several benefits, including fine-grained adaptation over 16 different levels for each different quality balance setting (compared with only three in the original BBN pipeline), as well as the ability to tailor other parameters, such as smoothness and latency, to meet user preferences or requirements.

Previous work on the Quasar video pipeline had assumed a stored media model in which video data would be preprocessed and the results stored on disk for (potentially repeated) streaming at a later time. This mode of operation is of little use in applications such as surveillance and remote monitoring where video data must be captured, processed, and transmitted in real-time. To overcome these limitations, we began the development of Qstream, a new version of the video pipeline that could support real-time transcoding of live video to the SPEG format.

We had originally planned to use a small autonomous robot vehicle, "OGImaBOT," as a demonstration platform for our work and for integration with the BBN OEP. This experiment was also chosen so that we could demonstrate the use of Timber in software for the robot that integrated both time-critical control, and rate-critical video processing tasks. The OGImaBOT featured a Pentium-based computer on the chassis of a RC truck, and was originally developed at OGI for use in a different project. After some initial experiments, however, we determined that OGImaBOT was not suitable for this work; aside from poor reliability and short battery life, the most serious problems were that it did not have enough memory to run the ACE, TAO, and QuO middleware

and that its CPU was too slow for real-time processing of live video. We therefore decided to build a new, low-cost robot vehicle, subsequently christened Timbot (the "Timber Robot"), using the same basic construction strategy as OGImaBOT, but with 256MB RAM, a more powerful Pentium III Processor running at 850MHz, a slot for a compact flash card/microdrive as backing store, and a small video camera on a pan-tilt mounting at the front of the robot. Construction of Timbot was completed by April 2002. By the time of the July 2002 PI meeting, we were able to use Timbot as an integrated part of our demonstration using the BBN OEP. In particular, we showed that we could successfully stream live data from the camera mounted on Timbot across a wireless network using our flexible, adaptive video pipeline software while simultaneously running simple robot control applications such as wall-following and obstacle avoidance, programmed in Timber, on the same CPU.

2.3 Project Year 3 (July 2002-June 2003)

Our efforts in the third year of the project focused almost exclusively on work in the context of the Boeing and BBN OEPs and on collaboration with researchers in other PCES projects. In the case of the Boeing OEP, we explored a new role for Timber in a domain-specific language for component configuration, and demonstrated significant benefits in terms of scalability, reuse, modularity, and defect detection/elimination. In the case of the BBN OEP, we continued to refine our work on adaptivity, overload tolerance, and graceful degradation, as well as working towards a coordinated technology demonstration with other PCES projects. The latter anticipated using Timbot and the OGI PPS software as a simulated UAV or UCAV, both in supplying video data to a remote ATR system, and in responding to high-level path control signals that could be used for target tracking. This work is described in more detail in the following sections.

2.3.1 A Domain Specific Language for Component Configuration

Prompted by discussions with members of the Boeing team at the PI meeting in April 2002, and by a follow-up visit by OGI personnel to the Boeing facility in St Louis, we began to explore a new application for Timber as the host of a new domain-specific language (DSL) for component configuration in the Boeing OEP. Although this particular application did not call for any of the

real-time or reactive features of the Timber language, we were able to leverage other features, particularly its support for powerful abstractions. In our initial experiments, we demonstrated that the resulting DSL would allow a significant reduction (by a factor of more than 30 in some of the larger cases) in the size of the descriptions for component configurations (and hence a significant improvement in scalability and productivity) in comparison to the XML-based approach that was used in the original Boeing OEP. We demonstrated that the DSL could support modular construction of large configurations, as would be required when the description of a large system is produced by a team of engineers instead of a single individual, and that it facilitated more effective reuse of previously built component subsystems. Moreover, in the process of generating new DSL versions of the configuration descriptions, we uncovered a number of previously undetected defects in the Boeing software ranging from minor typos and redundant code sections to broken invariants and typing errors. Because of the design that was used for the DSL, none of these errors could have occurred if the original descriptions had been constructed using the DSL. We also identified several inconsistencies between the informal prose specifications of some product scenarios and their representation in the XML format of the Boeing OEP. For example, in one instance, a component that was supposed to receive a 40Hz trigger signal had instead been configured with a 20Hz trigger. This problem was a little hard to spot in the original description because of a semantic gap between the specification and the implementation—one expressed this requirement in terms of frequency (in Hertz) while the other expressed it in terms of the reciprocal time interval (in microseconds). Although such mistakes could also have occurred using the DSL, it is reasonable to believe that they would be less likely because the DSL allows configurations to be expressed at a high-level, without requiring error prone, hand translation in cases like this.

These results provided strong indications of the benefits that domain-specific languages can provide in enhancing programmer productivity and software reliability. In addition, we found that the DSL enables significant new capabilities that are not easily supported by the original OEP. A small (but still useful) example of this was in providing a connection from DSL code to the dot format that is used by the AT&T GraphViz software, which allowed for visualization of component configurations in several different graphical formats.

In our initial experiments, we had reverse engineered the descriptions of several product scenarios in Build 1.6 of the Boeing OEP by hand to produce corresponding descriptions of those same scenarios in DSL code. These techniques were then quickly adapted to Build 2.0 of the Boeing OEP that was released at the beginning of the third project year. As we gained more experience in this, it became clear that the process could be automated, and we built an automatic XML to DSL reverse engineering tool to perform this task. This tool also incorporated static analysis and error checking functionality, which meant that it was immediately useful as a debugging aid to other developers working with the Boeing OEP, even if they were not interested in using the DSL code that the tool produced. The following table reflects the status of this work at the time of the PI meeting in December 2002, which was about half way through the third project year:

Scenario	#comps	XML (2.0)	XML (2.2)	DSL Hand	DSL RE	Factor
1.1	3	195	134	11	14	9.6
1.2	6	345	246	20	21	11.7
1.3	8	572	419	19	28	15.0
1.4	50	3,285	2,687	119	94	28.6
1.5	12	730	454	38	37	12.3
1.6	4	290	195	10	16	12.2
1.7	5	368	252	12	19	13.3
1.8	3	270	149	8	17	8.8
1.9	81	-	3,649	-	167	21.9
1.10	15	-	569	-	47	12.1
2.1	397	28,944	23,890	<660	819	29.2
3.1	4	240	145	-	19	7.6
3.2	7	392	279	-	27	10.3
3.3	17	1,005	627	-	47	13.3
3.4	91	-	4,566	-	196	23.3

This table summarizes details for the fifteen product scenarios that were included in Build 2.2, indicating the number of distinct components ("#comps") in each product scenario as well as the number of lines of XML code in the descriptions of these scenarios in Builds 2.0 and 2.2 of the OEP. (There are three gaps in the "XML (2.0)" column corresponding to three new product

scenarios that were only introduced in OEP Build 2.2.) The DSL columns show the number of lines of code in each of the corresponding DSL descriptions of these product scenarios. The "DSL Hand" column describes the DSL code examples that were produced by hand, while the "DSL RE" describes the DSL code examples that were generated automatically, and for all of the scenarios, by the reverse engineering tool. The "Factor" column indicates the ratio between the size of the XML description in Build 2.2 and the size of the corresponding, machine generated DSL code. A comparison between the "XML (2.0)" and "XML (2.2)" columns reflects a change that was made in the XML format used by the OEP to a more compact version that reduced size by 20-40%. Nevertheless, it is clear that the DSL still resulted in descriptions that are about an order of magnitude smaller, and that the benefits are greatest in the largest scenarios, which is notable because these scenarios are also the ones that are most representative of real systems.

Although we had provided detailed feedback to the OEP developers on the defects that we had found in previous releases of the OEP builds, we were still able to find a significant number of bugs in Build 2.2, most of which had occurred either as a result of the change in XML format or in the construction of the new product scenarios. These problems included deviations from the DTD (18 examples), a receptacle with an incorrect component type (1 example found), unused event suppliers (at least 25 examples found), and configurations of event suppliers and consumers using different port implementations (at least 72 examples found). Some of the problems that we found were of little practical consequence in the OEP as a whole (although they could have caused problems with interoperability or with a more rigorous implementation of the OEP framework), but others had the potential for unnecessary use of computational resources in support of redundant components or possible runtime failures in configured systems.

To assist other PCES researchers who were working on the Boeing OEP, we began to distribute our DSL implementation and tools on the web. One example of a collaboration that was assisted in part by this was a prototyped integration of our DSL with the Stanford Event Coordination Language (ECL).

2.3.2 Adaptivity, Overload Tolerance, and Graceful Degradation

Our earlier work had demonstrated the ability of the Qstream video pipeline and the PPS protocol to support dynamic adaptation over two quality dimensions (temporal and spatial) with sixteen different levels of adaptation for any given user preference/quality setting. In the third year of the project, we focused our efforts in this area on (i) exploring techniques for adaptation over more than two quality dimensions; (ii) generalizing the mechanisms of PPS and developing the underlying ideas as a pattern that could be applied in a wider range of applications; and (iii) packaging and documenting this work so that it could be more easily shared and transitioned into real world use.

We had already identified several different possibilities for exploring adaptation over new quality dimensions in both the robot control and video pipeline domains, but we decided that the latter would be more immediately appropriate for us given its relevance to the BBN OEP. We determined that it would be possible to accommodate both color space and region-of-interest (ROI) adaptation within our Qstream prototype. The former refers to the ability to vary the amount of color information that is transmitted with a video signal and was made feasible by the fact that the underlying SPEG format used in Qstream already separated color information into distinct components. The latter, ROI adaptation, refers to the ability to prioritize specified regions of a video signal more highly than others. This was also made feasible by our use of SPEG, which, being based on the MPEG1 standards, would allow us to represent regions in the video signal as collections of MPEG "macroblock" elements. We decided to focus on ROI adaptation first because we felt that it would provide a more demanding test of our methods, and that it would lead to a more interesting new capability.

The original system was written to support only two fixed dimensions of adaptation, so we began our work by building a new and more flexible mapper implementation for Qstream. (The mapper is the component of the PPS approach that implements QoS driven prioritization of the components in a scalable data format such as SPEG.)

During this year, we also began to investigate the role of utility functions—already used to map between objective and subjective notions of quality in Qstream—as a flexible way to specify QoS

policies for aspects of robot control such as power consumption, deviation from specified path, ability to detect obstacles, and speed. These were our first steps in generalizing PPS to a unified and flexible treatment of QoS adaptivity in both video and control aspects. Building on this, and also leveraging the new, more general mapper implementation, we extended the video pipeline to support simultaneous streaming of audio data. This work was helpful, not just in demonstrating the application of PPS to a new data formats, but also because it raised new challenges in its own right, including, most critically, the need for synchronization between multiple streams of data—audio and video signals in this particular case.

We pursued several other items of work in this area during the third project year including: the extension of the PPS protocol to support multicast, with a prototype implementation in Qstream; experiments to apply PPS to support streaming of still images, anticipating a potential role in demonstrations based on WSOA and on evolving plans for WSOA II; the development of a general version of the PPS protocol using Timber, accompanied by an increasingly general understanding of PPS as a design pattern for adaptive streaming; and the investigation of a role for Timber as a DSL for specifying sophisticated and flexible adaptation policies, in the style of the BBN CDL, for adaptation in multiple quality dimensions. We also produced our first public release of the Qstream software this year, which we made available on the web to other researchers both within and beyond the PCES program.

2.4 Project Year 4 (July 2003-December 2003)

In our final six months of technical work, we focused on three themes: Language middleware, exploring the applications of DSL technologies in real-time embedded systems design and implementation; Dynamic adaptation, developing reusable patterns for building flexible, QoS-sensitive applications; and Integration and transition, packaging our work to facilitate easier installation, use and integration/interoperability with other technologies, as well as demonstrating and documenting their benefits and applicability (and costs) for potential adopters.

The Boeing OEP continued to evolve, undergoing some major changes in infrastructure and file formats in the transition from Build 2.4 to Build 3.0, as well as adding new product scenarios. We

worked to ensure that our DSL implementation tracked successive releases of the Boeing software, and we continued to demonstrate similar benefits to those described previously,. The following table summarizes our status in this work in December 2003.

Scenario	#comps	XML LOC (3.0)	DSL LOC	Factor
1.1	3	130	13	10.0
1.2	6	247	20	12.3
1.3	8	411	27	15.2
1.4	50	2,671	93	28.7
1.5	13	479	38	12.6
1.6	4	184	15	12.3
1.7	5	241	18	13.4
1.8	3	141	16	8.8
1.9	81	3993	166	24.1
1.10	15	562	46	12.2
1.11	9	426	27	15.8
1.12	9	387	27	14.3
2.1	397	23,263	754	35.0
3.1	4	147	20	7.3
3.2	7	258	26	9.9
3.3	17	600	49	12.2
3.4	91	4280	213	20.1
3.5	3	129	20	6.4
4.1	572	31,796	1038	30.6

As before, this table demonstrates full coverage of all of the product scenarios in Build 3.0 with very similar reductions in code size as we had obtained in previous versions. Once again, although most of the bugs we had reported previously were fixed in either Build 2.4 or Build 3.0, we continued to uncover new bugs, including both XML errors and scenario configuration errors. For example, we discovered 80 components that were configured to generate events without corresponding listeners, and 8 instances of misconfigured master/proxy pairs such as proxies

without corresponding masters. None of these errors could occur with configurations that were generated using the DSL.

One of the changes introduced in Build 3.0 of the OEP was a requirement for new identifier tags in the XML code for event supplier and consumer definitions. This highlighted another benefit of the DSL approach because we were able to arrange for the DSL to XML translator to generate these identifiers automatically, without requiring any changes in the DSL descriptions of product scenarios. In the same way that high-level programming languages are used to produce code that is (largely) independent of the underlying machine, we were also able to use the DSL to gain some independence from details of the underlying, low-level XML format that was used in the OEP.

During this period, we began to investigate an enhancement to the reverse engineering tool that could automatically discover recurring structure in the large, monolithic configurations described in the OEP. Using a prototype, heuristic-driven tool, we were able to show that common patterns of component structure emerge in many places. The goals of this work included refactoring out complexity and identifying patterns for reuse so as to improve readability and usability of the DSL code produced by the reverse engineering tool. We also expected that this work would: (i) enable more efficient analysis of configurations; (ii) facilitate new forms of structural anomaly detection; and (iii) provide support for automated distribution of configurations across multiple processors. This last feature was considered to be particularly important because it could eliminate the significant and error prone burden of manual partitioning and because it could potentially be used to enable dynamic, automatic reconfiguration of a system in response to a processor failure.

We also worked to repackage DSL functionality as a standalone library, libdsl, that was designed to be used as a subproject in the OEP distribution (or indeed, in any other application where this might have been useful). In this way, we made it much easier for other researchers in the PCES program to leverage our DSL technology without the need to install or learn about additional tools (or even the DSL itself) before they could use DSL-based applications. Although this was originally conceived as a way to simplify use of the DSL as a design-time/compile-time tool, it soon became clear that this approach opened up new opportunities for use of libdsl as a run-time service. For example, libdsl provides a mechanism that could support dynamic reconfiguration of

components at run-time, which would constitute a significant new operational capability with the potential to increase mission agility, to enhance fault tolerance, to allow flexible load balancing, and to support fine-grained customization according to operator/preferences. Even though we did not attempt to optimize it in any way, the run-time footprint of our initial prototype implementation of libdsl was approximately 300KB, plus any additional space required for DSL scripts, which is smaller than we could expect to obtain using other approaches that incorporate a generic XML parser.

Another development involving our DSL was the construction of DTD-driven XML to DSL code generators supporting the ACL, ESCM, and AIF interface formats that were being used by other projects within the PCES program for design and analysis tools. This increased our opportunity for interoperability with other projects and with the tools that they were producing.

Work also continued on the theme of dynamic adaptation during the final six months of the project with continued focus on PPS as a strategy for adaptive streaming and on the refinement of the Qstream implementation. A new version of the Qstream software was released in November 2003 that included support for the multicast extension of PPS called PPM. We also worked to integrate the Qstream software with the AVstreams framework that runs on TAO. This was intended as a key step towards tighter integration of PPS into the BBN software so that it could be used in each stage of the OEP, not just from the UAV to the Distributor, and not just for video. The first proof-of-concept implementation of this functionality was completed in December 2003.

Technical work was stopped at the request of the sponsor at the end of December 2003. This early stop had not been anticipated, but we did our best to tidy up as many of the loose ends as we could. Inevitably, however, some technical lines of work that had been initiated at an earlier stage could not be completed within the scope of the project. Work on the Timber language and its compiler has been continued subsequently in a different context by former OGI personnel, but at a much slower rate of progress. There has not been a formal public release of the Timber compiler, but interim versions are available from publicly accessible cvs repositories. Work on Qstream has also continued under the lead of Dr. Buck Krasic who completed his thesis work in February 2004, part of which was supported by Project Timber. Krasic's dissertation documents the design of PPS as

well as the original implementation and experiments involving Qstream. Current versions of the Qstream software are available on the web from `http://qstream.org` and the emphasis has now switched to developing a software infrastructure for media streaming over the Internet. Work on ROI adaptation for Qstream had been suspended when the primary OGI researcher working on this topic took an extended maternity leave. She was about to resume work in this area when the project was stopped. Despite the promising results, we have not been able to continue work on the domain-specific language for component configuration in the Boeing OEP since the project was stopped because we no longer have access to OEP software releases. There is some hope, however, that ideas used in our DSL design might be transitioned to other domain-specific language designs, including a possible application to (meta-) models developed using the OMG Meta Object Facility (MOF).

3 Publications and Technical Paper Overview

Technical details of the work that has been carried out during this project are documented in a collection of papers and technical reports, most of which are included as appendices to this report. This section provides a brief overview of each of these documents and explains its relationship to the project work described in previous sections.

Real-time reactive programming for embedded controllers. R. B. Kieburtz, submitted to the ACM International Conference on Functional Programming, Florence, Italy, Sept. 3-5, 2001.

> [Included as Appendix A] This paper presents motivation and key elements of the Timed Reactive Systems (TRS) model on which the design of Timber was based.
>
> **Abstract:** Software-based controllers for physical devices and processes must provide both algorithmic functionality, which is the usual focus of computer programming, and timely management of events. Combining these essential aspects is the challenge of real-time programming. This paper takes a fresh look at the fundamental issues of this discipline and proposes a synthetic approach that can provide certain guarantees that temporal specications will be met.

Timber: A Programming Language for Real-Time Embedded Systems. Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz and Johan Nordlander, technical report, April 2002.

> [Included as Appendix B] This report describes the design of the Timber programming language including motivating examples, informal explanations of semantics, and details of concrete syntax.
>
> **From the introduction**: In this paper we provide a detailed but informal survey of Timber and its characteristic features. A formal semantic treatment of the language will appear in other papers; here the exposition will instead be based on short code examples. However,

we also introduce the semantic model that underlies one of Timber's main contributions: the way that time is integrated into the language.

Reactive Objects. Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black, Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002), Arlington, VA, 2002.

[Included as Appendix C] This short paper provides high-level insight into the motivation for the design of Timber and the advantages of programming with reactive objects.

Abstract: Object-oriented, concurrent, and event-based programming models provide a natural framework in which to express the behavior of distributed and embedded software systems. However, contemporary programming languages still base their I/O primitives on a model in which the environment is assumed to be centrally controlled and synchronous, and interactions with the environment carried out through blocking subroutine calls. The gap between this view and the natural asynchrony of the real world has made event-based programming a complex and error-prone activity, despite recent focus on event-based frameworks and middleware.

In this paper we present a consistent model of event-based concurrency, centered around the notion of reactive objects. This model relieves the object-oriented paradigm from the idea of transparent blocking, and naturally enforces reactivity and state consistency. We illustrate our point by a program example that offers substantial improvements in size and simplicity over a corresponding Java-based solution.

The Semantic Layers of Timber. Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. Springer-Verlag.

[Included as Appendix D] This paper presents a formal semantics for Timber that is structured in three distinct layers: functional, reactive, and scheduling. This work provides

a basis for reasoning about programs written in Timber as well as a starting point for language implementation.

Abstract: We present a three-layered semantics of Timber, a language designed for programming real-time systems in a reactive, object-oriented style. The innermost layer amounts to a traditional deterministic, pure, functional language, around which we formulate a middle layer of concurrent objects, in terms of a monadic transition semantics. The outermost layer, where the language is married to deadline-driven scheduling theory, is where we define message ordering and CPU allocation to actions. Our main contributions are a formalized notion of a time-constrained reaction, and a demonstration of how scheduling theory, process calculii, and the lambda calculus can be jointly applied to obtain a direct and succinct semantics of a complex, real-world programming language with well-defined real-time behavior.

Programming with Time-Constrained Reactions. Johan Nordlander, Magnus Carlsson, and Mark Jones. Submitted for publication, April 2004.

[Included as Appendix E] This paper explains the role of explicit timing constraints in Timber programs and the advantages that they provide in the development of real-time programming applications.

Abstract: In this paper we argue that a programming language for real-time systems should support the declaration of time constraints, and that those constraints should attach to a well-developed notion of reactions. To make our claims more precise, we introduce Timber, which is a concurrent programming language based on a model of non-blocking, reactive objects. Timber supports both upper and lower time constraints on a reaction, where an upper constraint corresponds to a classical deadline, and a lower constraint constitutes a very efficient way of scheduling an event to occur at a well-defined point in the future. A series of programming examples illustrates how these mechanisms can be used to express simple solutions to common problems in practical real-time programming.

Composed, and in Control: Programming the Timber Robot. Mark Jones, Magnus Carlsson, and Johan Nordlander, Technical Report, August, 2002.

[Included as Appendix F] This paper explains how the Timber programming language was used to provide a flexible and compositional approach to the development of control algorithms for Timbot, the Timber robot.

Abstract: This paper describes the implementation of control algorithms for a mobile robot vehicle using the programming language Timber, which offers a high-level, declarative approach to key aspects of embedded systems development such as real-time control, event handling, and concurrency. In particular, we show how Timber supports an elegant, compositional approach to program construction and reuse—from smaller control components to more complex, higher-level control applications—without exposing programmers to the subtle and error-prone world of explicit concurrency, scheduling, and synchronization.

Priority-Progress Streaming for Quality-Adaptive Multimedia. Buck Krasic and Jonathan Walpole, *In Proceedings of the ACM Multimedia Doctoral Symposium, Ottawa, Canada, October 2001.*

[Included as Appendix G] This short paper presents the key ideas underlying the design of the Priority Progress Streaming mechanism and its ability to support flexible, dynamic, and tailorable adaptation.

From the Abstract: We propose that streaming-media solutions targeted at the Internet must fully embrace the notion of graceful degradation, they must be architected with the expectation that they operate within a continuum of service levels, adjusting quality-resource trade-offs as necessary to achieve timeliness requirements. In the context of the Internet, the continuum of service levels spans across a number of time scales, ranging from sub-second timescales to timescales as long as months and years. We say sub-second timescales in relation to short-term dynamics such as network traffic and host workloads,

while timescales of months and years relate to the continuous deployment of improving network, compute and storage infrastructure.

We support our thesis with a proposal for a streaming model which we claim is simple enough to use end-to-end, yet expressive enough to tame the conflict between realtime and best-effort personalities of Internet streaming. The model is called Priority-Progress streaming. In this proposal we will describe the main features of Priority Progress streaming, which we have been implemented in a software-based streaming video system, called the Quasar pipeline.

Supporting Low-Latency TCP-Based Media Streams. Ashvin Goel, Buck Krasic, Kang Li, Jonathan Walpole, In Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002), Miami Beach, Florida, May 2002.

[Included as Appendix H] This paper describes modifications to standard implementations of the TCP protocol to provide improved support for low-latency streaming.

Abstract: The dominance of the TCP protocol on the Internet and its success in maintaining Internet stability has led to several TCP-based stored media-streaming approaches. The success of these approaches raises the question whether TCP can be used for low-latency streaming. Low latency streaming allows responsive control operations for media streaming and can make interactive applications feasible. We examined adapting the TCP send buffer size based on TCP's congestion window to reduce application perceived network latency. Our results show that this simple idea significantly improves the number of packets that can be delivered within 200 ms and 500 ms thresholds.

Infopipes: An Abstraction for Multimedia Streaming. Andrew Black, Rainer Koster, Jie Huang, Jonathan Walpole, and Calton Pu, Multimedia Systems (special issue on Multimedia Middleware), 8(5), pp. 406-419, ACM / Springer-Verlag, 2002.

[Included as Appendix I] This paper describes a high-level abstraction for describing and building a wide range of distributed streaming applications, including quality adaptive systems or components such as Qstream as special cases.

Abstract: To simplify the task of building distributed streaming applications, we propose a new abstraction for information flow—Infopipes. Infopipes make information flow primary, not an auxiliary mechanism that is hidden away. Systems are built by connecting pre-defined component Infopipes such as sources, sinks, buffers, filters, broadcasting pipes, and multiplexing pipes. The goal of Infopipes is not to hide communication, like an RPC system, but to reify it: to represent communication explicitly as objects that the program can interrogate and manipulate. Moreover, these objects represent communication in application-level terms, not in terms of network or process implementation.

Adaptive Live Video Streaming by Priority Drop. Jie Huang, Charles Krasic, Jonathan Walpole, and Wu Chi Feng, IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS 2003), Miami, FL, July 2003.

[Included as Appendix J] This paper describes the techniques used to extend PPS to support live video streaming in the Qstream implementation, including experimental results.

Abstract: In this paper we explore the use of Priority-Progress Streaming (PPS) for video surveillance applications. PPS is an adaptive streaming technique for the delivery of continuous media over variable bit-rate channels. It is based on the simple idea of reordering media components within a time window into priority order before transmission. The main concern when using PPS for live video streaming is the time delay introduced by reordering. In this paper we describe how PPS can be extended to support live streaming and show that the delay inherent in the approach can be tuned to satisfy a wide range of latency constraints while supporting fine-grain adaptation.

Quality-Adaptive Media Streaming by Priority Drop. Charles Krasic, Jonathan Walpole, Wu-chi Feng, in Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2003), Monterey, California, June 2003.

[Included as Appendix K] This paper presents a thorough overview of the PPS protocol as a mechanism for supporting overload tolerance and graceful degradation.

Abstract: This paper presents a general design strategy for streaming media applications in best effort computing and networking environments. Our target application is video on demand using personal computers and the Internet. In this scenario, where resource reservations and admission control mechanisms are not generally available, effective streaming must be able to adapt in a responsive and graceful manner. The design strategy we propose is based on a single simple idea, priority data dropping, or priority drop for short. We evaluate the efficacy of priority drop as an adaptation tool in the video and networking domains. Our technical contribution with respect to video is to show how to express adaptation policies and how to do priority-mapping, an automatic translation from adaptation policies to priority assignments on the basic units of video. For the networking domain, we present priority-progress streaming, a real-time best-effort streaming protocol. We have implemented and released a prototype video streaming system that incorporates priority-drop video, priority mapping, and priority-progress streaming. Our system demonstrates a simple encode once, stream anywhere model where a single video source can be streamed across a wide range of network bandwidths, on networks saturated with competing traffic, all the while maintaining real-time performance and gracefully adapting quality.

A Framework for Quality Adaptive Media Streaming. Charles C. Krasic. Ph.D thesis. OGI School of Science & Engineering at OHSU, February 2004.

This Ph.D. dissertation provides comprehensive treatment of PPS, including motivation, high-level design, implementation in the Qstream prototype, and empirical evaluation. For

reasons of space, we do not include the full document as an appendix but note that it is available online from http://www.cs.ubc.ca/~krasic/publications/krasic-phd.pdf.

Abstract: This dissertation presents a general design strategy for streaming media applications in best effort computing and networking environments. Our target application scenario is video streaming using commodity computers and the Internet. In this scenario, where resource reservations and admission control mechanisms are generally not available, effective streaming should be able to adapt to variations in bandwidth in a responsive and graceful manner. The design strategy we propose is based on a single simple idea, adaptation by priority data dropping, or priority drop for short. We evaluate the efficacy of priority drop in the video and networking domains.

For video, we show how common compression formats can be extended to support priority drop, thereby becoming streaming friendly. In particular, we demonstrate that priority-drop video allows adaptation over a wide range of rates and with fine granularity, and that the adaptation is tailorable through declarative adaptation-policy specifications. Our main technical contribution is to show how to express adaptation policies and how to do priority-mapping, an automatic translation from adaptation policies to priority assignments on the basic units of video.

In the networking component of this thesis, we present two versions of Priority-Progress Streaming, a real-time best-effort streaming protocol. The basic version does classic unicast streaming for video on demand style streaming applications. The extended version supports efficient broadcast style streaming, through a multi-rate multicast overlay.

We have implemented a prototype video streaming system that combines priority-drop video, priority mapping, and the Priority-Progress Streaming protocols. The system demonstrates the following advantages of our approach: a) it maintains timeliness of the stream in the face of rate fluctuations in the network, b) it utilizes available bandwidth fully thereby maximizing the average video quality, c) it starts video display quickly after the user initiates the stream, and d) it limits the number of quality changes that occur. In

summary, we will show that priority-drop is very effective: a single video source can be streamed across a wide range of network bandwidths, and on networks saturated with competing traffic, all the while maintaining real-time performance and gracefully adapting quality.

A Domain Specific Language for Component Configuration. Mark P Jones. Internal Draft for distribution to PCES TDs, May 8, 2002.

[Included as Appendix L] This report was prepared for use by PCES TDs and collaborators to document results from the initial development of the Timber-based DSL for the Boeing OEP. The report was not submitted for approval to be released or published in any venue with a more general audience.

Introduction: This note describes a domain specific language (DSL) for component configuration in the Boeing Open Experimental Platform (OEP). The purpose of the DSL is to make it easier for system integrators to construct and validate configurations from concise, modular, and reusable high-level descriptions. The design of the DSL reflects common patterns, terminology, and notations used in the specific domain, which in this case have to do with initializing software components, and establishing connections between them. By providing direct support for domain specific idioms, a DSL empowers its users to express their ideas quickly and concisely, to work more productively, to avoid certain kinds of coding error, and to tackle more complex problems than might otherwise be possible. The DSL that we describe here, for instance, has been used to produce a clear and modular description of the largest example in the current OEP build that is approximately 25 times smaller than the original description written in XML.

A Domain Specific Language for Component Configuration: Preliminary User Notes. Mark P. Jones. Documentation distributed with releases of the DSL software. Summer 2003.

[Included as Appendix M] Although this document was prepared as user documentation, it also provides details on the design, structure, and benefits of the DSL as it had matured towards the end of the project.

4 Conclusions

This report documents the work and results of the project "Temporally Aware Reactive Systems," also referred to informally as "Project Timber." The original goals of this project were to develop new language technology and analysis tools to support a compositional approach to programming of real-time embedded systems.

Responding to corresponding changes in focus within the PCES program as a whole, the technical focus of our work changed considerably during the lifetime of the project. At a high-level, however, the project successfully and consistently pursued two main themes:

- The application of domain specific language technology to increase programmer productivity and to improve software reliability, as demonstrated in a wide range of applications spanning hard real-time control of a robot vehicle, specification of dynamic adaptation policies, and build-time configuration of large component-based software systems.

- The development and application of general patterns and protocols for building flexible, QoS-sensitive applications that can adapt dynamically and automatically to changes in their environment and in the computational resources that are available to optimize performance with respect to user-specified quality preferences.

In both of these areas, working in the context of both Open Experimental Platforms as well as with independent examples, we were able to demonstrate significant benefits and capability enhancements over the state of the art as represented, for example, by the two OEPs. These technologies have matured during our work on this project to the point that they can now realistically be considered for use in the construction of new, operational systems.

APPENDIX A

Real-time reactive programming for embedded controllers. R. B. Kieburtz, submitted to the ACM International Conference on Functional Programming, Florence, Italy, Sept. 3-5, 2001.

Real-time Reactive Programming for Embedded Controllers *

Richard B. Kieburtz
Oregon Graduate Institute

ABSTRACT

Software-based controllers for physical devices and processes must provide both algorithmic functionality, which is the usual focus of computer programming, and timely management of events. Combining these essential aspects is the challenge of real-time programming. This paper takes a fresh look at the fundamental issues of this discipline and proposes a synthetic approach that can provide certain guarantees that temporal specifications will be met.

1. EMBEDDED REAL-TIME, REACTIVE SOFTWARE

Many researchers during the past several years have explored ways in which functional programming languages, extended with monads to permit disciplined uses of effects, can be applied to programming problems that have formerly been considered to require the use of lower level, imperative languages. In virtually every case in which this paradigm has been tested, it has resulted in simpler, clearer programs that benefit from the power of abstraction, expressive type systems and conciseness of functional language notations. This paper takes extended functional programming into a new domain, that of embedded, reactive software with explicit time constraints.

Embedded software has become ubiquitous in the devices, appliances and vehicles in common use today. The term refers to software whose role is to define the behavior of a host system that incorporates physical devices, operates continuously in real time and does not require direct human control or interaction. In an increasing number of applications, an embedded software controller acts as a surrogate for a human as, for instance, in the anti-lock braking system of an automobile.

Embedded software can be *active*, when it defines the temporal behavior of a process, or *reactive*, when it responds to signals from external processes. The software controlling a digital music synthesizer is primarily active; the software controlling an anti-lock braking system is primarlily reactive. This paper is concerned with the temporal aspects of reactive embedded software.

A surprising fact about the accepted design notations for so-called real-time software is that time is rarely mentioned, or at least is not mentioned explicitly. The terminologies more commonly used to specify temporal behaviors are *rates* of periodic processes and *priorities* of aperiodic ones. This paper explores the consequences of directly specifying the temporal constraints imposed by a host system on its software controller, with the motivation that the control required of the program might be deduced from such a specification. The apparently challenging task of programming real-time responses can be made declarative!

A reactive program responds to input events from a host system by producing output events that are transmitted as signals to the host. Its responses are effected by programmed *reactions*; code sequences that are functional in the sense that we understand functions to be interpreted in monads that account for effects. We think of reactions as functions from *State* to *State*, interpreted in the *Events* monad. *O'Haskell* [9] is a programming language particularly well-suited for writing reactive programs.

The control of a reactive program determines which reactions shall be invoked in response to anticipated input events. This control could be described in imperative programming style, as it is in the *Esterel* language [2], or in a functional style, as it is in *Lustre* [4]. In this paper we shall show how the control for a real-time reactive program can be derived from a specification of the temporal constraints on its behavior.

Section 2 discusses general characteristics of embedded, reactive software. Time constraints are introduced in Section 3. In Section 4 we outline an algorithm for computing a schedule that assures that declared time constraints will be met. Section 5 describes objects in *O'Haskell* and argues why its semantics are appropriate to real-time reactive programs. An example is presented in Section 6. Section 7 discusses the integration of computationally demanding, critical-rate tasks with time-critical tasks. The paper concludes with a discussion of related work.

2. CHARACTERISTICS OF REAL-TIME REACTIVE SYSTEMS

In a typical real-time application, the task of an embedded software system is to react to signals generated by a host

*The research reported here has been supported by DARPA contract numbers F33615-98-C-3516 and F33615-00-C-3042.

system in which it resides, providing output signals to effect control of the host system. The host may be a physical system, equipped with sensors of time-varying parameters of its state, and with actuators that operate motors, valves or other physical control devices. A host system might also include communications hardware, real-time databases, or other time-varying data sources.

In such an application, the embedded software is not only required to deliver functionality, but to simulate a process that is loosely synchronized in time with external processes in its host system. Physical processes in the host system are considered to evolve in continuous time. The time constraints imposed by a host system add a challenge to the task of programming embedded software.

A specification for a real-time, reactive, software controller results from analyzing the host system in which the controller is to be embedded. Interactions between the controller and the host system occur through discrete events. An *input event* is a discrete, atomic message from the host system to the controller. Each occurrence of an input event is registered when it arrives at the controller. Registration of an input event implies that the fact of its occurrence has been recorded in a register that is readable by the controller. If a data word accompanies an input event, we shall assume it is latched in a designated register that can be read by the software.

A software controller reacts to occurrences of input events by emitting output events to its host system. The first task in analyzing the requirements for a controller is to determine what input events it must respond to and what output events may be expected in response to each of these inputs. The next task is to determine timing constraints that relate these events.

Both the expected responses to input events and the timing constraints may depend upon the *mode of operation* of the host system. For example, the operational mode of an aircraft when taxiing on the ground at low speed is very different from the mode of the same aircraft in flight. When taxiing, inputs from airspeed and altitude measurement sensors require no response and can be ignored, while inputs from wheel rotation sensors are significant. An input from the pilot's control yoke may be interpreted differently (i.e. may evoke different responses from the controller) when an aircraft is taxiing than when it is airborne. Furthermore, the time constraints imposed to achieve stable control might be expected to be more lax when an aircraft is taxiing than when it is in flight.

2.1 Hybrid systems

A *hybrid* system design combines the aspects of a continuous state space with a finite set of operating modes that partition the state space. The hybrid system concept is widely used to organize and simplify the design of controls for highly nonlinear systems. In a hybrid design, the control laws used to regulate or guide a system may be switched as the system's state trajectory transitions between modes of operation.

From the vantage point of the control engineer, the operating modes of a dynamical system are regions of its state space in which the system dynamics may be approximated by a simplified system. The simplified dynamics is only expected to provide an accurate approximation within a defined region of the system's state space. When operating in different modes, the system state is subject to different sets of constraints, and consequently, may satisfy a different set of dynamical equations.

The state and control signals in a hybrid system may have both continuous and discrete aspects. A quantity that varies continuously with time may be modeled by a time series of discrete values. A discrete event can be localized in time. A discrete event may be said to occur in a time interval during which the (continuous) trajectory of a state variable crosses a designated threshold, as, for instance, when the wheels of an aircraft touch the ground in landing. Control signals may also have discrete aspects, as when a motor is turned on, or a braking actuator is applied.

From the vantage point of a software engineer who is contemplating the design of an embedded software controller, its operating modes correspond to discrete states of a finite-state transition system. A control state is characterized by two maps:

1. A mapping from anticipated input events to programmed *reactions*. A reaction may emit output events and update the internal state variables of the controller.

2. A mapping of input events to control-state *transitions*, which may be guarded by boolean predicates over the internal variables.

In this view of a controller as a hybrid system, the controller weakly simulates the controlled system, tracking its mode transitions with transitions of the controller's own control states, and tracking the trajectory of the system's state within modes through discrete changes in the controller's internal state variables.

2.2 Control states and guarded reactions

Reactive programs are often designed as finite transition systems. Although the systems we consider may mutable state variables, this notion of state is not the one we wish to consider as discrete, because the space of valuations of the state variables may be infinitary or even continuous. Rather, we wish to consider a finite partition of this space, which we shall refer to as the set of *control states* of the system.

A control state is characterized by a mapping of input events to finite sets of guarded reactions. A *guarded reaction* is prefaced by a boolean predicate over the state variable. The meaning of a guarded reaction is that if its guard evaluates *True*, the reaction is enabled, otherwise it is not. A control state is deterministic if at most one guard on its set of guarded reactions can be *True* for any valuation of the state variables, otherwise it is nondeterministic. When a set of guarded reaction is dispatched in response to an input event, its guards are evaluated, and if any reaction becomes enabled, one of the enabled reactions is selected for invocation.

Since a reactive program is of bounded length, it can contain only a finite number of guarded reactions, hence, there is only a finite number of control states, no matter what may be the cardinality of valuations state variables. Since executing a reaction may alter the values of state variables, it can induce a transition of the control state, i.e. change the values of some guards.

Guards must be restricted so as not to contain calls to recursively defined functions, so the evaluation time of a guard can be bounded. Guards are pure expressions, i.e. they can be interpreted in the *Identity* monad.

2.2.1 Hierarchical transition systems and concurrent objects

A reactive system is often designed as a collection of interacting component subsystems, which operate concurrently and interact asynchronously by exchanging messages. A system designed in this way can be modeled as a system of nested, nondeterministic, finite automata [5]. It can be implemented as a system of concurrent objects with asynchronous methods to support interaction.

In a concurrent object implementation, a reactive system has a composite state represented by the internal, private state variables of its component objects. The composite state of the system is distributed over the states of its components. Reactions are realized by method invocations. Distinct objects may execute methods concurrently, corresponding to concurrent transitions of independent transition systems. As objects have private state variables, methods of distinct objects share no common variables and hence do not interfere when executed concurrently.

2.3 Synchronizing a controller with external processes

The "real-time" aspect of an embedded software controller is the requirement that it must be synchronized with one or more external processes in its host system. The precision needed in this synchronization is calibrated by the time delay that can be tolerated between the arrival of an input event and the delivery of an output event in response to the input. If synchronization is not sufficiently precise, the accuracy of information represented by a series of input events will be degraded, as well as the accuracy of the control effected by a series of output events. Extreme degradation of control accuracy because of excessively delayed response can result in loss of stability of the dynamical system under control.

The precision required to synchronize a controller with its host system depends upon the nature of the interaction and may not be uniform over all classes of events, even within a single application. We shall call the allowable time delay between an input and an output the *response time* of the input-output pair. The required response times for events in a system can be represented by a partial function $response\_time : Inputs \times Outputs \rightarrow Time$.

When there are multiple channels for interaction between a controller and its host system or when the host system is comprised of several independent processes, multiple input events may arrive simultaneously at the controller. In physical terms, simultaneity means that the separation in arrival times of two or more events is less than the temporal precision with which event arrivals are registered.

When input events cannot be separated by their time of arrival, it makes no sense to schedule their responses in order of arrival. Instead, the specified response times for input-output pairs provides a scheduling criterion. A criterion for a successful schedule of responses is that regardless of the order in which input events may have arrived, the response to every input event is emitted within the required response time.

2.4 Process velocity and inter-event latencies

Another factor influencing the design of a software controller is the rate or rates at which external processes in the host system are able to deliver input events to the controller. The anticipated event arrival rates, multiplied by the per event cost of processing, determine the process performance required of a controller.

Event arrival rates are not necessarily uniform, as some classes of events may occur at unpredictable times. Nevertheless, there is inevitably a latent period separating the generation of an event by a process from the generation of a subsequent event by the same physical process. Informally, we refer to the rate at which a process is capable of generating event occurrences as its *process velocity*. A reciprocal measure is the *minimum inter-event latency* of the process.

Since we do not ordinarily wish to specify in detail the external processes that constitute the host system of a software controller, we shall instead specify inter-event latencies directly. The latencies characterizing the maximum velocities of external processes can be specified by defining a partial function, $latency :: (Inputs \cup Outputs) \times Inputs \rightarrow Time$.

2.4.1 Clocks

Periodic sampling of a continuous process provides a uniform process velocity as observed by interactions with a control process. The apparent velocity of a sampled process is determined by the frequency of a clock process that strobes sensors of its observable parameters. A clock is necessarily a process external to a software controller, because the software has no inherent access to the timing provided by the hardware processor that executes it.

We distinguish three commonly used modes of interaction with an external clock process:

- a *synchronizing* clock delivers a stream of primitive input events (ticks) to which a software controller reacts. A tick event carries no data, but ticks may be accumulated in an internal state variable as a measure of elapsed time. The tick of a synchronizing clock provides an event at which a software controller can be activated to react to any input events that have arrived since the preceding clock tick.

- a *metric* clock does not deliver input events, but provides a read-only register that can be read at any time to provide a measure of "absolute" time. This allows the controller to measure elapsed time by taking the difference between a current and a prior reading of the metric clock register.

- an *interval timer* is a clock that can be set by an output event emitted by the controller. A timer-setting event specifies a time interval after which the clock will respond with an input event.

A real-time operating system normally provides a timing service that incorporates one or more of these modes of clock interaction.

2.5 Reactions take time

In defining the *synchrony assumption* in connection with reactive programming, Berry and Gonthier state that "reactions are presumed to be instantaneous" [2]. This slogan represents the assumption that the computation time needed for a controller to calculate a reaction to a given input event is very much less that the allowed response time to this or any other event. Insofar as this assumption is valid, the output events emitted by a reaction to an input event occur synchronously with the input event.

Refining this notion slightly, the idea of synchrony can be tied to the precision with which the time of occurrence of events can be discriminated, by any observer in a system. If an observer can discriminate events a and b whenever they are separated by an interval of at least Δt, then the response time for a reaction that responds to a by emitting b must be less than Δt, for otherwise, the system's temporal specification would be sufficiently lax that an observer might detect the delay introduced by its processing of events.

Clearly, the assumption of negligible reaction times is not valid in all circumstances. In developing a discipline of real-time reactive programming, we are interested in situations in which reaction times must be accounted for. We shall relax the assumption of instantaneous reactions to one that seems more realistic. We assume that for each reaction, there is a stable distribution of its execution times, which can be estimated by measurements conducted on the computing platform on which an embedded software system is to be installed. We assume that these distributions have Gaussian tails and therefore allow us to predict upper bounds on the execution times of individual reactions that can be expected to hold with probability as near to 1 as is required.

Let $ReactionTime :: Reactions \rightarrow Time$, where $Time$ is a type synonym for the positive real numbers. $ReactionTime\rho$ is an upper bound on the time taken for a synchronous execution of the reaction ρ.

Unlike response time and latency, reaction times cannot be obtained by analysis of requirements imposed by the host system. Reaction times can be determined only after a software controller has been designed and implemented. One way to determine reaction times is by measuring the time taken for repeated executions of reaction codes on a platform nearly identical to that on which the controller is to be installed. From the measurements, one can determine a distribution of times for each reaction. The distributions provide an empirical basis from which to estimate upper bounds on the execution times of individual reactions.

2.5.1 Reactions are atomic actions

Reactions are functions (interpreted in the *Event* monad[1]) that are programmed to implement the functional requirements of an embedded controller. A reaction is applied to the current state of the controller in response to an input event. The computation of a reaction must occur atomically and without noticeable interruption of processor service, for otherwise, reaction times could not be predicted with any degree of confidence. Furthermore, the code body of a reaction must not contain calls to any recursively defined function, nor can it invoke other reactions, because its execution time must be uniformly bounded. Viewed as procedural code, the body of a reaction can have no loops.

However, the scheduling of reactions need not be programmed as it would be in an imperative reactive language. Time-bounded reactions can be dispatched in response to arrivals of input events according to fixed schedule. If there exists any feasible schedule that can assure that execution of time-bounded reactions can always satisfy the time constraints of a temporal system specification, then such a schedule can be calculated statically from the system specification and the bounds established on reaction times. In Section 4 we present an algorithm for calculating a feasible static schedule.

[1]The *Event* monad is a composition of a state monad with a monad whose effects are discrete interactions with a host system. The command **emit** [*event*] is a non-proper morphism of the *Event* monad. The state object of the monad is internal to the program and is not visible to the host system.

3. TIME-CONSTRAINED EVENT SYSTEMS

Taking a more abstract view of real-time systems, we shall designate as an *event* any interaction between a program and the environment in which it runs. An event may carry a value or not. An event class is designated by an identifier, optionally bound to a type. Let

$Events$	be a finite set of events,
$Inputs \subseteq Events$	be a finite set of input events,
$Outputs \subseteq Events$	be a finite set of ouput events,

This partition of events is arbitrary, but useful in conceptualizing the interactions of a program with its environment.

3.1 Specifying temporal properties

The host system of an embedded controller determines temporal constraints on the interactions of the controller with the host system. A temporal system specification binds identifiers to the possible input and output events, gives the type of data (if any) that accompanies an event, and prescribes constraints on the temporal orderings of events.

$Inputs :: identifier \mid \ldots \mid identifier$
$Outputs :: identifier \mid \ldots \mid identifier$
$ResponseTime :: (Inputs, Outputs) \rightarrow Time$
$DelayedResponse :: (Inputs \cup Outputs, Outputs) \rightarrow Time$
$Latency :: (Inputs \cup Outputs, Inputs) \rightarrow Time$

The constraints that define temporally correct behavior of the control process are interpreted as follows.

- If $ResponseTime(a, b)$ is defined and yields the time interval $\delta_{a,b}$, then whenever an occurrence of event a evokes emission of event b in response, event b must be emitted no later than $\delta_{a,b}$ seconds after event a has arrived. Response times characterize a reactive process.
- If $DelayedResponse(a, b)$ is defined and yields the time interval $\Delta_{a,b}$, then whenever an occurrence of event a evokes emission of event b in response, event b may occur no earlier than $\delta_{a,b}$ seconds after the occurrence of event a. Delayed responses characterize an active process.
- If $Latency(a, b)$ is defined and yields the time interval $\gamma_{a,b}$, then we can be assured that an occurrence of input event b will never happen earlier than $\gamma_{a,b}$ seconds following an occurrence of event a.

An unconstrained input event may potentially occur at any time. Therefore, we expect that a complete temporal system specification will include a latency relation for every possible input event. In particular, it is quite common to have a latency specification of the form $Latency(a, a) = \delta_a$, which prescribes the minimum time in which an input event a can follow a previous occurrence of the same sort of event.

A temporal system specification must be accompanied by a functional description of a host system, which may be

given formally or informally. A functional description specifies the modes of operation of the host system and determines what outputs should be emitted in reponse to possible inputs in each of the operating modes.

3.1.1 Satisfying response time constraints

The discrete event hypothesis arose because of limited ability to resolve occurrences of events in continuous time. Events whose arrival times cannot be resolved are considered to have occurred simultaneously.

As a simplifying assumption, suppose that the reaction to a set of two or more input events can be any sequential composition of the reactions to the individual events, as if they had arrived in an arbitrary order. A sufficient condition for a system of synchronous reactions to satisfy the response time constraints of a system is met if

a. the latency relation forbids the occurrence of an unbounded number of events in any finite time interval, and

b. for any set of events whose latency constraints do not preclude their simultaneous occurrence, the sum of reaction times is less than the specified response time for any reaction to an event in the set.

However, the above condition may be far more stringent than necessary, for it fails to take into account our ability to schedule the dispatch of reactions to a set of input events that await responses.

4. INFERRING A FEASIBLE SCHEDULE FROM REAL-TIME CONSTRAINTS

Given a finite set of enabled reactions, each with a predictable execution time and a deadline time by which each reaction is to be completed, an *earliest-deadline first* (EDF) algorithm [8] will determine a feasible schedule, if one exists.

This easy scheduling problem is complicated by the arrivals of new input events that enable additional reactions. We must frame the algorithm in relative time, rather than absolute deadline times, because the response times allowed for reactions are measured from the time of arrival of events. We must also take into account the latencies that constrain the arrival of input events, for otherwise, a system may be overloaded with an unbounded number of input events that require timely responses.

New events may arrive from the host system while the controller is dormant or while it is executing a reaction to a previously registered event. We assume that the controller is activated immediately if a new event should arrive while it is dormant. (This might be accomplished by preempting the execution of a non-real-time task.) When a reaction completes executing, the controller polls registers in its environment to determine if new events have arrived. If events have arrived, the controller's dispatch function determines which reaction should be the next to execute.

We shall consider the problem of calculating a static schedule, or dispatch table, for each control state of the controller. A schedule is calculated from the temporal specification of events required by the host system and the execution time bounds for reactions.

Since the latencies constraining the arrivals of input events depend upon the history of activity (both input events recently arrived and output events recently emitted), discovering a feasible schedule has aspects of a dynamic programming problem. We calculate a schedule over a bipartite tree, whose nodes are labeled by configurations of time-constrained event sets.

A *scheduling configuration* is represented as a record of type:

$$\begin{array}{l}\{state :: State;\\ \ evts :: \{(Inputs \times Int)\};\\ \ constrained :: \{(Inputs \times Int)\}\}\end{array}$$

where *state* represents the simulated control state, *evts* is a set of input events whose arrival has been registered with the controller, each paired with a bound on the (simulated) time since its arrival, and *constrained* is a set of latency-constrained events, each paired with a lower bound on the time remaining until an event of this sort might arrive. Transitions of the control state are determined by a state transition function, $transition : State \times Reaction \rightarrow State$.

A configuration, C, is said to be *feasible* if it satisfies the following condition. For each pair (e, t) in *evts*, let the associated reaction, when the controller state is s, be denoted by $\rho_{s,e}$ (notice that this reaction might be *null*). Let $\omega_{s,e}$ designate the set of output events that might be emitted by $\rho_{s,e}$. Let $\tau_{s,e} = min_{e' \in \omega_{s,e}} RT(e, e')$.

The feasibility condition for a configuration is:

$$\begin{array}{rl} feasible(C) = & \forall s : State \, \forall (e,t) \in C.evts \bullet \\ & (\tau_{s,e} - t - ReactionTime\, \rho_{s,e} \geq 0) \\ & \vee \, (\rho_{s,e} = null) \end{array}$$

The calculation of a schedule elaborates a branching-time, temporal simulation of the possible interactions of the controller and its host system. This simulation can be modeled as a finitely branching tree of unbounded depth.

A scheduling tree contains two types of nodes, which alternate along any path descending from the root. The node types are:

- *dispatch* nodes. Arcs emanating from a dispatch node are labeled by reactions.
- *arrival* nodes. Arcs emanating from an arrival node are labeled by sets of input events.

A scheduling tree is like a game tree. At a dispatch node, any outward arc may be chosen to continue elaboration. At an arrival node, no choice is allowed. The simulation must be elaborated along all outward arcs. The root of a scheduling tree is labeled by an arrival configuration that specifies the initial control state, an empty set of registered events, and an empty set of constraints on arriving events.

To elaborate an arc from an arrival node labeled by C and whose incident arc is labeled by reaction ρ, construct outgoing arcs labeled by members of the powerset, $\wp(Inputs - \{c' \mid (c', t) \in C.constrained\})$. The configuration labeling the node reached by an arc whose label is σ is:

$$\begin{array}{l}\{state = C.state;\\ \ evts = C.evts \cup \{(e,t) \mid e \in \sigma \wedge t = ReactionTime\, \rho\};\\ \ constrained = C.constrained\}\end{array}$$

A newly arrived event is pessimistically assumed to have arrived at the time the preceding reaction was invoked, thus when execution of the reaction concludes, the event has already been resident for the duration of the reaction time.

To elaborate an arc from a dispatch node labeled by feasible configuration C, choose $(e,t) \in C.evts$, if $C.evts$ is not empty. Let $\rho_e = reactionTo(C.state, e)$. Form a new node labeled by the configuration,

$$\begin{aligned} C' = \{&state = transition(C.state, \rho_e);\\ &evts = \{(e',t') \mid (e',t) \in C.evts\\ &\qquad \wedge\, e' \neq e\\ &\qquad \wedge\, t' = t + ReactionTime\,\rho_e\};\\ &constrained = \{(c,t') \mid (c,t) \in C.constrained\\ &\qquad \wedge\, t' = t - ReactionTime\,\rho_e\\ &\qquad \wedge\, t' \geq 0\}\} \end{aligned}$$

Label an arc from node C to C' with the reaction ρ_e. If a node label is not a feasible configuration, then that node has no successors in the tree.

A scheduling tree is feasible if at every dispatch node, C, either $C.evts \neq [\,]$, or C has at least one feasible successor.

4.1 Configuration subsumption

The purpose of elaborating paths in a scheduling tree is to discover and eliminate paths that inevitably lead to infeasible configurations. These, along with the paths that terminate on dispatch nodes with empty *evts* sets, are the finite paths in the tree. Clearly, we must have a criterion for stopping the elaboration of a path when we discover that it is not finite.

Algorithm termination is based upon the notion of configuration subsumption. Informally, we say that a configuration C subsumes a configuration C' if for every feasible path elaborated from C', there is a corresponding path that can be elaborated from C. The notion of correspondence of two paths is that they have the same sequence of edge labels.

More precisely, we say that C subsumes C' if the following conditions are satisfied:

$$\begin{aligned} &C.state = C'.state\\ &\forall (e',t') \in C'.evts\ \exists (e,t) \in C.evts \bullet e = e' \wedge t \geq t'\\ &\forall (c,t) \in C.constrained\ \exists (c',t') \in C'.constrained \bullet\\ &\qquad\qquad e = e' \wedge t \leq t' \end{aligned}$$

That is, C' has the same state as does C, every event registered in C' is also registered in C with possibly longer time of residence in C than in C', and every event constrained in C is also constrained in C' with its time of constraint in C possibly shorter than in C'.

4.2 A static scheduling algorithm

We have implemented a scheduling algorithm in Haskell which elaborates a scheduling tree as outlined above. Its result is either a schedule, represented by a set of quadruples of type $(State \times \wp(Inputs) \times Inputs \times Reactions)$, or it reports failure. A quadruple $(s, evts, e, r)$ is interpreted by the dispatch function of a controller to mean "when in state s with a set of registered input events *evts*, dispatch reaction r and retire event e".

The algorithm returns only a single feasible schedule, although there may be many. If the choice of a reaction to choose in elaborating a dispatch node of the scheduling tree does not result in a feasible schedule, other choices must be explored for completeness. The algorithm limits its choices to the EDF strategy. Although this strategy is not complete for the scheduling problem we have posed, it seems that it would require a pathological construction to demonstrate an example on which EDF fails, yet a feasible schedule exists.

Configuration subsumption limits the depth of exploration of a scheduling tree. The algorithm maintains two lists of configurations whose feasibility has been resolved: those from which a feasible, infinite path from a dispatch node labeled with the configuration in known to exist and those for which it is known that there is no such feasible path. In addition, it maintains a list of the unresolved configurations of dispatch nodes on the path from the node being explored back to the root.

Whenever the current node's configuration subsumes a previously successful configuration or the unresolved configuration of an ancestor node, the algorithm ceases to elaborate its successors. Either the configuration has been proven to be successful, leading to at least the same feasible paths as the successful configuration that it subsumes, or the algorithm has discovered a path segment that can be repeated indefinitely, implying that no new information will result from its further exploration.

Whenever the configuration of the current node is subsumed by that of a node that generates only infeasible paths, it has been proven to be unsuccessful, for either it mandates more stringent deadline constraints or it allows unconstrained events to enter sooner than does the configuration that subsumes it.

The lazy evaluation of Haskell limits the amount of computation necessary to find a feasible schedule. The algorithm simulates backtracking by producing a list of the feasible schedules that can be discovered from each choice point [11]. These lists are, of course, evaluated lazily, and often only the first element is demanded. The strategy of saving both the feasible and infeasible configurations that have been encountered in elaborating a scheduling tree effectively gives the algorithm a dynamic programming behavior rather than that of a less efficient, backtracking algorithm.

5. ASYNCHRONOUS REACTIVE OBJECTS IN O'HASKELL

Components of a reactive system designed as a hierarchy of interacting, finite-state transition systems are quite naturally programmed as objects with a particular set of characteristics. Each component automaton can be represented by an object. Objects used in this way have a number of characteristics:

- *Independence*—distinct components do not share state;
- *Concurrency*—actions of distinct objects may execute concurrently with respect to one another;
- *Asynchrony*—inter-component communication normally occurs through non-blocking messages;
- *Sequentiality*—the actions of an individual component occur in sequence, not interleaved;
- *Passivity*—a component is activated only by the messages to which it reacts; it has no background activity;

Since objects do not share state, components are coupled only through their message-passing interfaces. This allows their actions to be executed concurrently without additional synchronization. An asynchronous message triggers the execution of a method in an object, but the sender of a message does not await a return. The sender of a message and the

actor which receives the message and reacts to it are not synchronized.

Since the actions (method executions) of an object do not block on messages they may send, each action, once initiated, can be executed to completion. An object executes its actions sequentially, without interruption. Furthermore, an object is active only when it is executing one of its methods in response to a message it has received.

This set of characteristics (one might call them restrictions on an object's behavior) is characteristic of the *actor* model of computation [1]. It has been used as a basis for the semantics of objects in *O'Haskell* [9], a functional, object-oriented programming language.

5.1 O'Haskell—a functional language with simple objects

O'Haskell embeds a purely functional expression language based upon Haskell version 1.3, complementing the expression language with objects that encapsulate state. Unlike conventional OO languages, *O'Haskell* does not support object inheritance, in which a hierarchy of objects can share components of state with other objects above them in the hierarchy. It does support subtyping, allowing a programmer to declare a hierarchy of object types. The methods of *O'Haskell* objects are first-class entities, as are the objects themselves. The monadic type system inherited from Haskell eliminates the possibility of confusion between statements and expressions. We have used O'Haskell for prototyping reactive systems programs.

5.1.1 O'Haskell has no explicit threads

One of the most taxing aspects of concurrent programming in a conventional OO language, such as Java, is synchronizing the activities of multiple threads. In Java, threads are explicitly created by a program. Multiple threads can be active in an object and thus can potentially share simultaneous access to the object's state. To ensure deterministic behavior for an object in which multiple threads can run, thread activities must be synchronized with monitors that control access critical regions of code in which state variables may be accessed.

In O'Haskell, threads are implicit, rather than explicit. Exactly one thread is allocated (implicitly) for each object. That thread is awakened when the object executes a method in response to a received message. The thread is dormant when there are no outstanding messages for the object.

5.1.2 Synchronous and asynchronous methods

A method of an O'Haskell object may be either a synchronous or an asynchronous action. A synchronous *request* returns a value to the sender of a message calling the request. The sender waits for the result. An asynchronous *action* is non-blocking on the sender of a message. The computational consequences of an asynchronous action can be seen in the update of an object's state, or in the receipt (by other objects) of additional messages that may be sent in executing the action.

Requests and actions are typed in the *Object* monad in the O'Haskell type system, so that they cannot be confused with pure functions. (The *Object* monad may be considered to subsume the *IO* monad of Haskell.) An method typed in the *Object* monad can reference the state variables of the object and can be composed of commands that include destructive assignments to object variables, as well as sending messages to objects. O'Haskell provides a convenient **do** syntax that can be used in the code body of a method to compose sequential commands.

6. EXAMPLE: THE REFLEX GAME

As an illustration, we shall consider the *Reflex Game*, a simple reactive system that has been widely studied as an example problem for reactive programming languages [3, 7]. Here, however, we shall explicitly specify temporal constraints on the game.

In the Reflex Game, a human player tests the speed of her reflexes by pressing a button as quickly as possible after given a signal to do so. The controller measures the elapsed time between display of the go-ahead signal to the player, and sensing the button push. To deter anticipatory responses by the player, the controller delays display of the go-ahead signal by a randomly selected interval following the player's signal that she is ready to play. An early push of the button by the player terminates the game. If the player pushes the *Ready* button when it is not expected, a warning bell is sounded. A new game is initiated by the player by depositing a coin in a slot. A game consists of a fixed number of trials.

Following this informal description of the game, we specify the input and output events of the controller:

```
Input Events
   Coin,  Ready,  Stop

Output Events
   Game_light_on, Game_light_off,
   Warning_bell, Go_ahead_light_on
   Go_ahead_light_off, Tilt_light_on,
   Tilt_light_off, Display (Int)
```

The response time constraints impose maximum times (in milliseconds) between an input event and an output event delivered in response to that input. Latencies provide minimum intervals that separate an input event from a specified sort of preceding input (or output) event.

```
Response Times
   (Coin,Game_light_on,250),
   (Ready,Warning,150),
   (Stop,Go_ahead_light_off,10),
   (Stop,Warning_bell,150),
   (Stop,Game_light_off,150),
   (Stop,Tilt_light_on,150)

Latencies
   (Coin,Coin,2000),
   (Ready,Ready, 1000),
   (Stop,Stop,1000)
```

Notice that in the set of temporal constraints declared above, the output event *Go_ahead_light_off* delivered in response to a press of the *Stop* button has a much shorter response time than any of the other events. We shall return to this issue when we discuss the calculation of a schedule.

One more specification is needed to govern the temporal behavior of the system. Delay times specify minimum intervals that separate output events when there are no intervening arrivals of input events. Delay times regulate the

rates at which spontaneous generation of outputs can occur. In the specifications given below, two sorts of spontaneous output events are regulated. The `Game_light_off` event, occurring spontaneously, ends a game because there has been no input from the player for a predetermined timeout interval. The `Display` event may occur spontaneously to refresh the display at the beginning of a new trial. The delay specified between occurrences of the `Display` event ensures that if the user does not initiate a new trial herself, the display will hold the value of her reflex time for a specified interval.

```
Delays
   (Go_ahead_light_on,Game_light_off,10000),
   (Game_light_on,Game_light_off,10000),
   (Warning_bell,Game_light_off,10000),
   (Display,Display,3000),
   (Display,Game_light_off,3000)
```

6.1 The Reflex Game as a hybrid system

Although the controller for the Reflex Game is not required to simulate the dynamical behavior of a continuous system, it must nevertheless maintain an approximate record of the passage of time, in addition to other, discrete state variables. From the point of view of the designer, a hybrid system is one in which there is a finite space of control states that is much smaller than the state space of valuations of the controller's state variables.

For the Reflex Game, we have designed a five-state controller. States *Idle, Holding, Counting, Play* and *Display* correspond to the operating modes informally described above. A state transition diagram is shown in Figure 1.

From the diagram in Figure 1, the reader will notice that the system makes transitions guarded by temporal events—when the elapsed time in a state exceeds a given bound—in addition to transitions triggered by input events. The bounds on elapsed time spent in states *Play, Counting* and *Display* are extracted from the delay declarations given in the preceding section. For instance, a transition from state *Play* to *Idle* can occur 10000 msec. after a transition into the *Play* state, if no input-induced transition has occurred. This is determined from the delay constraint between an occurrence of output event `Game_light_on`, which is emitted by the reaction to a `Coin` drop event, and an occurrence of `Game_light_off`, which is emitted by a reaction that accompanies entry into the `Idle` state. The bound on elapsed time in state *Holding* is a value calculated from a pseudo-random sequence generator.

6.2 Scheduling reactions of the Reflex Game

Specifications of input and output events, response time constraints, delay times and latencies are obtained from the problem specification, The set of control states, state transitions, the set of reactions and the output events emitted by each reaction cannot be gotten by specification alone but may be extracted from the program of a controller by a straightforward analysis. The execution times of reactions cannot be determined accurately by analysis, but a distribution can be measured. The combined data are gathered in a text file for input to the scheduling algorithm.

For the five-state controller of the Reflex Game, with ten reactions and estimated reaction times ranging from 10 to 35 ms., the scheduler produces a scheduling table with 76 entries. A typical entry reads as:

```
(Counting,[Coin,Stop],Stop,DisplayTime)
```

Its interpretation is that when the controller is in state *Counting* and the set of registered events is {*Coin,Stop*}, then the *DisplayTime* reaction is to be dispatched, retiring the event *Stop*.

The existence of a feasible schedule is sensitive to the time constraints and the estimated reaction times given to the scheduler. When the scheduler reports that there is no feasible schedule, a designer has several options. Obviously, it can be important to sharpen estimates of the bounds on reaction times. However, redesign of the finite-state transition system that defines the controller can also improve schedulability.

For instance, when the controller is in state *Counting* (refer to Figure 1) the *Stop* event causes a reaction that emits a *display* output, which has a critical response time relative to the arrival of the *Stop* input. However, if a *Ready* event were received just before a *Stop* event occurred, the *Ready* event would be reacted to first, and would leave the controller still in the *Counting* state. In the worst case, accounted for by the scheduler, the execution time for a reaction to this hypothetical *Ready* event must be subtracted from the response time to obtain the time remaining for the reaction to the *Stop* event to be completed. It is this sort of circumstance that is detected by the static scheduling algorithm when it reports no feasible schedule.

A designer can sometimes remedy such a circumstance by state-splitting in the design of the controller. In the case outlined in the preceding paragraph, a new state, *Reset*, might be introduced. If a *Ready* event occurred when the controller was in the *Counting* state, it would transition to the *Reset* state, from which the *Counting* state would be reentered after another random time interval. Any intervening *Stop* event that occurred while in the *Reset* state could be ignored. The possibility of temporal competition between reactions to the *Stop* and *Ready* events in the *Counting* state would be eliminated. Obviously, any such redesign must satisfy the requirements of the application.

6.3 A statically scheduled simulation of the Reflex Game in O'Haskell

To program a simulation of the Reflex Game in *O'Haskell*, we have declared a module called `game` to represent the controller. This module takes as parameters the imported environment that provides access to actuators linked to physical control of the game devices, a statically calculated schedule for dispatching reactions to events (Section 4) and the seed of a random sequence generator. For simulation, the module imports a `Tk` environment instead of an actual machine to realize the game interface.

The first entity declared in the body of the `game` module is an object template that declares a set of state variables, whose scope is restricted to methods of the `game` object.

```
game env schedule seed =
   template  -- the game state and initial values
      state      := Idle
      evts       := Empty
      time       := 0 :: Int
      rand       := 0 :: Int
      startTime  := undefined
      totalTime  := 0 :: Int
      trialNumber:= 0 :: Int
```

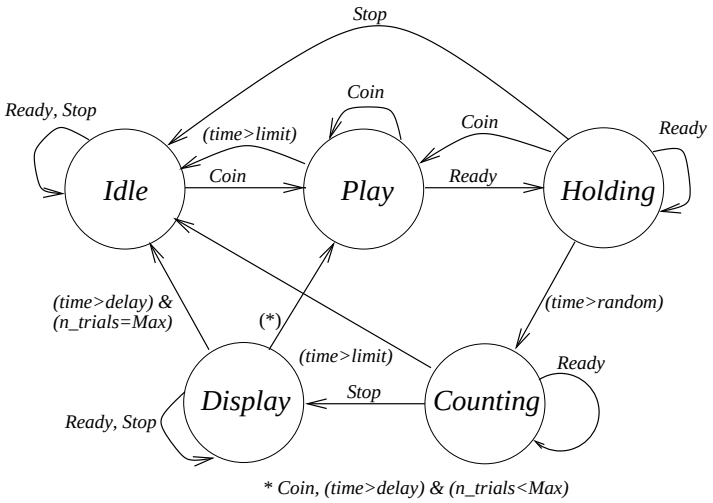


Figure 1: Control states of the Reflex Game

The `game` module also includes declarations of `action` methods that define the programmed reactions of the controller to input events. Since the reactions to input events may be particular to each control state, there are more reactions defined than the number of possible input events. Our controller defines ten reactions, including a "no-op" reaction for completeness.

Since the game undergoes state transitions on temporally defined conditions, it is useful to distinguish a sort of internal event that we choose to call *Tick*. The *Tick* event will be calibrated by one of the timing services provided by the environment.

A typical reaction is `count`, which responds to a *Tick* event when the controller is in the *Holding* state.

```
count = action
   if time >= delayTime then
      st <- env.timeOfDay
      startTime := st
      setGameState Counting Go_light_on
   else
      time := 1 + time
      setTimer
```

The guard on this reaction is manifested in an **if** ... **then** ... **else** clause. On a *True* value of the guard, a call to `setGameState` realizes a control state transition and simultaneously emits an output event. On a *False* value of the guard, there is neither a state transition nor a visible output event, but a state variable (`time`) is updated and a call to `setTimer` resets an interval timer in the environment.

In addition to the code of its ten reactions, the `game` module contains

- a `main` procedure that initializes the display environment,
- an `arrival` action that responds to the arrival of input events by adding new arrivals to the set of outstanding inputs that await reactions,
- the methods `setGameState` and setTimer,
- a dispatch method that interprets the `schedule` argument and reads the `evts` variable. The dispatch method runs a reaction whenever the set of outstanding events is non-empty and removes from this set the event to which the reaction responds.

Control of reactions is not explicitly programmed . The Reflex Game controller is a declarative realization of a time-constrained state transition system.

6.4 Timing

In the Reflex Game simulation, we have used a low-resolution interval timer provided by the `Tk` interface to calibrate the delays specified for the controller. The `setTimer` action requests a *Tick* event in 500 ms. It is unnecessary, in this application, to calibrate delays with any greater precision.

Where more precise timing is needed, the Reflex Game implementation requests a reading from a metric, time-of-day register provided by its run-time environment. This relies ultimately on the timing of the hardware platform on which the application runs. The player's reaction time is taken to be the difference of two readings of this clock. The first reading is taken by a reaction to a *Tick* event, when the "Go" light is turned on. The second reading is taken by a reaction to a *Stop* event caused when the player presses a button. The response times declared for these events determine bounds on the precision expected of this measurement.

The Reflex Game as programmed in *Esterel* [3] relies upon a synchronizing clock with 1ms. precision. This imposes a burdensome performance requirement on the computing system that supports the game controller, for it must respond to tick events of this clock at a considerably higher frequency than other events are expected to occur in the system.

Contrast that situation with the version of the example proposed here. In our version, a metric clock measures time rather than using the software controller to count the ticks of a synchronizing clock. An interval timer emits tick events at most every 500 ms, rather than at every millisecond interval. The average rate of arrival of events in this version will be far lower than the rate imposed by tick events in the version timed by a synchronizing clock.

In general, reducing the arrival rate of time-critical events is a good thing, for it reduces the frequency of expensive context switches, allowing more processing time to be dedicated to tasks that do not have firm response time constraints. Although the Reflex Game does not specify any such tasks, many complex embedded applications do involve computation-intensive tasks that rely upon processor cycles to be available between reactions to time-critical events.

7. CRITICAL-RATE TASKS

Up to this point, the focus of our attention has been time-critical events, whose response times are typically far shorter than inter-event latencies. Reactions to these events must be very short computations, for reactions must execute within the required response times.

Many applications also entail tasks that require more computation time, but whose responses are less urgent. We call these *critical rate* tasks, for the rates at which they provide service is more critical than the actual times of delivery of the service.

For example, a controller of an autonomous vehicle that employs terrain-following guidance will execute time-critical tasks that directly control the vehicle to maintain its course and attitude. In addition, there will be a critical-rate task that responds to terrain-sensing inputs such as radars, sonars, or computer vision systems, and which may consult a geographical database to navigate the vehicle. The output of such a task will be a course heading to follow during the next segment of the vehicle's trajectory. The navigation task requires far more processor time than does vehicle control. Its response time is dictated by the vehicle's speed and the lengths of segments for which headings are calculated. This response time may be as long as the latency between requests for a new course heading.

Operationally, one distinction between time-critical and critical-rate tasks is that the former must be executed without interruption to assure response times, whereas the latter must be assured a specified fraction of the total processor time during each reaction. Conventionally, rate-monotonic scheduling is used to determine whether the processing needed by a set of critical-rate tasks can be provided.

7.1 Interactions between time-critical and critical-rate tasks

In the example of a terrain-following autonomously controlled vehicle discussed above, the vehicle control task holds the vehicle on course for an individual path segment. Path segments are calculated incrementally, as the vehicle proceeds along its trajectory, by the navigation task. The heading and length of the next segment must be communicated to the vehicle control task by the time the vehicle reaches the end of the segment it is traversing.

In the O'Haskell object model, we can distinguish the objects that serve time-critical and critical-rate tasks by priorities. The methods of an object given the priority to react to time-critical events are assured to gain the processor immediately when enabled by an input event, and to execute without interruption. They can therefore be scheduled statically to assure that the response times declared for time-critical tasks will be met.

The methods of an object given the lower priority of a critical-rate task execute when no time-critical task is enabled. Execution of these methods can be interrupted to execute a time-critical task. The O'Haskell object model guarantees that these tasks cannot interfere in their access to variables, as the state variables are strictly partitioned by objects.

Communication between objects of different priority is handled by asynchronous method calls that pass parameters as values, i.e. by copying the parameter to a register or a temporarily allocated buffer, rather than by passing references. When a method executing at time-critical priority invokes a method of an object with critical-rate priority, the invocation request the method to run after the higher priority task has completed. In the reverse direction, when a method of a critical-rate object invokes a method of a higher priority, time-critical object, the method invocation is an input event to the time-critical object. It immediately preempts the rate-critical object to react to the event. In this asymmetric message-passing protocal, locks are not required for synchronization and a static schedule guarantees that data are never lost.

8. CONCLUSIONS

When the real time response requirements of an embedded, reactive software system are explicitly specified and bounds on the execution times of its reactions are known, a schedule for dispatching atomically executed reactions can be calculated statically. We have prototyped a static scheduling algorithm in Haskell and demonstrated its use with an example. Advantages of this approach with respect to techniques in common use are

- When a static schedule is based upon time constraints, it assures that a specified synchronization is achieved between a software controller and the host system in which it is embedded. When scheduling is based upon priorities alone, the time of responses cannot be assured.
- Statically scheduled reactions demand use of the processor only when one or more input events have actually arrived, demanding responses. It accommodates the service of events whose arrival times are relatively unpredictable with less context-switching overhead than does a uniformly clocked, event-loop design.
- When reactions are implemented as methods of single-threaded objects, per-object sequentiality guarantees that reactions do not interfere in their access of state variables, while methods of separate objects can execute concurrently without interference. Thread scheduling is replaced by static scheduling of reactions. In contrast, if multiple threads are allowed to enter an object, explicit monitors are needed to avoid possible interference of multiple threads accessing the state variables of an object.

8.1 Related work

The research most closely related to the results presented in this paper is the *Rialto* project [6], which supports programming of time-critical tasks with declarations of time constraints that are used by a static scheduling algorithm to allocate time intervals in a CPU schedule. This interesting project has slightly different goals than ours. Its primary emphasis is CPU reservations in an operating system, while ours is the design of event-driven embedded systems.

Objective Caml [10] embeds and concurrency primitives into a functional programming language, as does *O'Haskell*. However, there are significant differences. In *Ocaml*, state, objects and concurrency are orthogonal aspects. They can be used in conjunction or independently. In *O'Haskell*, these language aspects have been integrated to provide a disciplined way to construct concurrent, imperative programs. The *O'Haskell* discipline is well suited to the approach taken in this paper to the design of reactive systems. A second difference between the two languages is that *Ocaml* is strict, while expression evaluation in *O'Haskell* is lazy. That difference, however, is not critically important to the style of programming discussed in this paper.

Acknowledgement: The author is indebted to his colleagues on the *Timber* project, Mark Jones, Johan Nordlander and Magnus Carlsson, for their encouragement, contributions of suggestions and ideas, and especially for the design and implementation of *Hugs* and *O'Haskell*, the versatile software tools used in this research.

9. REFERENCES

[1] Gul A. Agha. *Actors : A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1987.

[2] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[3] F. Boussinot. Programming a reflex game in Esterel v3_2. Rapport de recherche 07/91, Centre de Mathématiques Appliquées, Ecole des Mines de Paris, Sophia-Antipolis, 1991.

[4] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[5] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] Michael B. Jones, Daniela Roşu, and Marcel-Cătălin Roşu. CPU reservationd and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 198–211. ACM Press, October 1997.

[7] Richard B. Kieburtz. Reactive functional programming. In D. Gries and W-P. de Roever, editors, *Programming Concepts and Methods (PROCOMET'98)*. Chapman-Hall, June 1998.

[8] C. L. Liu and J. W. Leyland. Scheduling algorithms for multiprogramming in a hard, real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[9] Johan Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Chalmers University of Technology, 1999.

[10] Didier R
'emy and Jérôme Vouillon. Objective ML: an effective, object-oriented extension to ML. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[11] Philip Wadler. How to replace failure by a list of successes. In *2'nd International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France, September 1985. Springer-Verlag.

APPENDIX B

Timber: A Programming Language for Real-Time Embedded Systems. Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz and Johan Nordlander, technical report, April 2002.

Timber: A Programming Language for Real-Time Embedded Systems

Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz and Johan Nordlander
timber@cse.ogi.edu

Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
Beaverton, Oregon, USA

In this paper we provide a detailed but informal survey of Timber and its characteristic features. A formal semantic treatment of the language will appear in other papers; here the exposition will instead be based on short code examples. However, we also introduce the semantic model that underlies one of Timber's main contributions: the way that time is integrated into the language.

May of the features of Timber have been adopted from the reactive object-oriented concurrent functional language O'Haskell [15], which was in turn defined as an extension to the purely functional language Haskell [12]. However, the Haskellian ancestry of Timber should not cause it to be ignored by the wider (non-functional) programming language community. Indeed, Timber attempts to combine the best features of three different programming paradigms.

- Timber is an **imperative object-oriented language**, offering state encapsulation, objects with identity, extensible interface hierarchies with subtyping, and the usual complement of imperative commands such as loops and assignment. Inheritance in the style of, *e.g.*, Smalltalk, is not presently supported, but this is an area that we continue to study. The lack of inheritance is largely counterbalanced by rich facilities for parameterization over functions, methods, and templates for objects. Additional Timber features not commonly found in object-oriented languages include parametric polymorphism, type inference, a straightforward concurrency semantics, and a powerful expression sub-language that permits unrestricted equational reasoning.
- Timber can also be characterized as a strongly typed **concurrent language**, based on a monitor-like construct with implicit mutual exclusion, and a message-passing metaphor offering both synchronous and asynchronous communication. However, unlike most concurrency models, a Timber process is represented as an object, that is, as the unit of state encapsulation. Moreover, execution of a process should not be regarded as continuous, but should instead be thought of as consisting of a

sequence of **reactions** to external events (made visible to the object as messages). These reactions always run to completion (*i.e.*, they are non-blocking) and in mutual exclusion. The execution order of reactions is determined by baselines and deadlines associated with these events. When we speak of reactive objects in the sequel, this is what we mean.

- Timber is finally a **purely functional language** that supports stateful objects through the use of monads. Timber allows recursive definitions, higher-order functions, algebraic datatypes, pattern-matching, and Hindley/Milner-style polymorphism. Timber also supports type constructor classes and overloading as in Haskell, but these features are not central and the Timber extensions to Haskell do not depend on them. To this base Timber conservatively adds two major features: *subtyping*, and a monadic implementation of stateful *reactive objects*. The subtyping extension is defined for records as well as datatypes, and is supported by a powerful partial type inference algorithm that preserves the types of all programs typeable in Haskell. The monadic object extension is intended as a replacement for Haskell's standard IO model, and provides concurrent objects and assignable state variables while still maintaining referential transparency.

The exposition here largely follows the informal survey of O'Haskell in Nordlander's thesis [15]. Section 1 presents a brief overview of the base language Haskell and its syntax, before we introduce the major type system additions of Timber: *records* and *subtyping* (Sections 2 and 3). In Section 4 our approach to *type inference* in Timber is presented. The rôle of *time* is introduced in Section 6. *Reactive objects*, *concurrency*, and *encapsulated state* are discussed in Section 5. Section 7 presents some additional syntactic features of Timber, before the paper ends with an example of Timber programming (Section 8). The grammar of Timber appears in the Appendix.

1. Haskell

Haskell [1, 12] is a lazy, purely functional language, and the base upon which Timber is built. Readers familiar with Haskell may wish to skip this section; it introduces no new material, and is present to make this paper accessible to those who have not previously met the language, or who need a reminder of its features and syntax.

Functions

Functions are the central concept in Haskell. Applying a function to its arguments is written as a simple juxtaposition; that is, if f is a function taking three integer arguments, then

```
f 7 13 0
```

is an expression denoting the result of evaluating f applied to the arguments 7, 13, and 0. If an argument itself is a non-atomic expression, parentheses must be used as delimiters, as in

```
f 7 (g 55) 0
```

Operators like + (addition) and == (test for equality) are also functions, but are written between their first two arguments. An ordinary function application always binds more tightly than an operator, thus

```
a b + c d
```

should actually be read as

```
(a b) + (c d)
```

Laziness

The epithet *lazy* means that the arguments to a function are evaluated only when absolutely necessary. So, even if

```
g 55
```

is a non-terminating or erroneous computation (including, for example, an attempt to divide by zero), the computation

```
f 7 (g 55) 0
```

might succeed in Haskell, if f happens to be a function that ignores its second argument whenever the first argument is 7. This kind of flexibility can be very useful for encoding and manipulating infinite data structures, and for building functions that play the rôle of control structures.

One of the consequences of laziness is that it can sometimes become quite hard to predict when computation will actually take place, and calculating worst case execution times is correspondingly difficult. Whether the costs of laziness outweigh the benefits in a language intended for real-time programming is an open question, and one that we will continue to examine experimentally. It is important to note that none of the extensions to Haskell that we put forward in Timber relies on laziness. Thus it is perfectly reasonable to judge the merits of our extensions as if they were intended for an eager programming language, and it would be perfectly possible to give Timber an eager semantics without major surgery.

Function definitions

Functions can be defined by equations on the top-level of a program. They can also be defined locally within an expression. The following fragment defines the function f at the top-level; the function sq is defined locally within the body of f.

```
f x y z = let sq i = i * i
          in sq x * sq y * sq z
```

Note that the symbol = denotes *definitional* equality in Haskell (*i.e.*, = is neither an assignment nor an equality test). Local definition of a function within other definitions is also possible, as in

```
f x y z = sq x * sq y * sq z where
      sq v = v * v
```

Anonymous functions can be introduced with the so-called *lambda-expression*, written using the symbols \ …-> in lieu of λ…. So

```
\ x y z -> x*y*z
```

is an expression whose value is the function that multiplies together its three arguments. An identical function is defined and named product by the definition

```
product x y z = x*y*z
```

The scope of a name can be limited by a **let** expression, so

```
let product x y z = x*y*z in product
```

has as its value the same anonymous function as the original lambda expression.

Type inference

When introducing a new variable, the programmer does not in general have to declare its type. Instead, the Hindley-Milner-style type inference algorithm employed in Haskell is able to discover the most general type for each expression. This often results in the inference of a *polymorphic type, i.e.,* a type expression that includes one or more variables standing for arbitrary types.

The simplest example of a polymorphic type is that inferred for the identity function,

```
id x = x
```

The most general type that can be ascribed to the function id is a -> a: this type is polymorphic, since a is treated as if it were universally quantified, that is, “for all types a”. However, the programmer can also use an explicit type annotation to indicate a more specific type, as in

```
iid :: Int -> Int
iid x = x
```

Partial application

A function like f above that takes three integer arguments and delivers an integer result has the type

```
Int -> Int -> Int -> Int
```

Arrow associates to the right, so this means Int -> (Int -> (Int -> Int))

Hence such a function need not always be supplied with exactly three arguments. Instead, functions can be *partially applied*; a function applied to fewer than its full complement of arguments is treated as denoting an anonymous function, which in turn is applicable to the missing arguments. This means that (f 7) is a valid expression of type Int -> Int -> Int, and that (f 7 13) denotes a function of type Int -> Int. Note that this treatment is consistent with parsing an expression like f 7 13 0 as (((f 7) 13) 0).

Pattern-matching

Haskell functions are often defined by a sequence of equations that *pattern-match* on their arguments, as in the following example:

```
fac 0 = 1
fac n = n * fac (n-1)
```

which is equivalent to the more conventional definition

```
fac n = if n==0 then 1 else n * fac (n-1)
```

Pattern-matching using Boolean *guard* expressions is also available, although this form is a bit contrived in this simple example.

```
fac n    | n==0        = 1
         | otherwise   = n * fac (n-1)
```

Moreover, explicit **case** expressions are also available in Haskell, as shown in this fourth variant of the factorial function:

```
fac n = case n of
          0  ->  1
          m  ->  m * fac (m-1)
```

Algebraic datatypes

User-defined types (called *algebraic datatypes* in Haskell) can be defined using data definitions, which define a kind of labeled union type with name equality and recursive scope. Here is an example of a data definition for binary trees: it declares three identifiers, BTree, Leaf and Node.

```
data BTree a   =   Leaf a
               |   Node (BTree a) (BTree a)
```

The type argument a is used to make the BTree polymorphic in the contents of its leaves; thus a binary tree of integers has the type BTree Int. The identifiers Leaf and Node are called the *constructors* of the datatype. Constructors, which have global scope in Haskell, can be used both as functions and in patterns, as the following example illustrates:

```
swap (Leaf a) = Leaf a
swap (Node l r) = Node (swap r) (swap l)
```

This function (of type BTree a -> BTree a) takes any binary tree and returns a mirror image of the tree obtained by recursively swapping its left and right branches.

Predefined types

In addition to the integers, Haskell's primitive types include characters (Char) as well as floating-point numbers (Float and Double). The type of Boolean values (Bool) is predefined, but is an ordinary algebraic datatype. Lists and tuples are also essentially predefined datatypes, but they are supported by some special syntax. The empty list is

written [], and a non-empty list with head x and tail xs is written x:xs. A list known in its entirety can be expressed as [x1,x2,x3], or equivalently x1:x2:x3:[]. Moreover, a pair of elements a and b is written (a,b), and a triple also containing c is written (a,b,c), *etc*.

As an illustration of these issues, here is a function which "zips" two lists into a list of pairs:

```
zip (a:as) (b:bs)    = (a,b) : zip as bs
zip _      _         = [] .
```

Note that the order of these equations is significant.

The names of the types of lists and tuples are analogous to the terms: [a] is the type of lists containing elements of type a, and (a,b) denotes the type of pairs formed by elements of types a and b. Thus the type of the function zip above is [a] -> [b] -> [(a,b)]. There is also degenerate tuple type (), called *unit*, which contains only the single element (), also called unit.

Strings are just lists of characters in Haskell, although conventional string syntax can also be used for constant strings, with "abc" being equivalent to ['a','b','c']. The type name String is just a *type abbreviation*, defined as:

```
type String = [Char]
```

String concatenation is an instance of general list concatenation in Haskell, for which there exists a standard operator ++, defined as

```
[] ++ bs = bs
(a:as) ++ bs = a : (as ++ bs)
```

Haskell also provides a primitive type Array, with an indexing operator ! and an "update" operator //. However, this type suffers from the fact that updates must be implemented in a purely functional way, which often means creating a fresh copy of an array each time it is modified. We will see later in this paper how monads and stateful objects enable us to support the Array type in a more intuitive, as well as a more efficient, manner.

Higher-order functions

Functions are first-class values in Haskell, so it is quite common for a function to take another function as a parameter; such a function is known as a higher-order function. map is a typical example of a higher-order function; map takes two arguments, a function and a list, and returns a new list created by applying the function to each element of the old list. map is defined as follows:

```
map f [] = []
map f (x:xs) = f x : map f x
```

The fact that map is higher-order is exposed in its type,

```
map :: (a -> b) -> [a] -> [b]
```

where the parentheses are essential. As an example of how map can be used, we construct an upper-casing function for strings by defining

```
upCase = map toUpper
```

where toUpper :: Char -> Char is a predefined function that capitalizes characters. The type of upCase must accordingly be

```
upCase :: [Char] -> [Char]
```

or, equivalently,

```
upCase :: String -> String  .
```

Layout

Haskell makes extensive use of indentation — two dimensional layout of text on the page — to convey information that would otherwise have to be supplied using delimiters. We have been using this convention in the foregoing examples, and will continue to do so. The intended meaning should be obvious. It is occasionally convenient to override the layout rules with a more explicit syntax, so it may be good to keep in mind that the two-dimensional code fragment

```
let   f x y = e1
      g i j = e2
in g
```

is actually a syntactic shorthand for

```
let { f x y = e1 ; g i j = e2 } in g   .
```

Informally stated, the braces and semicolons are inserted as follows. The layout rule takes effect whenever the open brace is omitted after certain keywords, such as **where**, **let**, **do**, **record** and **of**. When this happens, the indentation of the next lexeme (whether or not on a new line) is remembered and the omitted open brace is inserted. For each subsequent line, if it contains only whitespace or is indented more, then the previous item is continued (nothing is inserted); if it is indented the same amount, then a new item begins (a semicolon is inserted); and if it is indented less, then the layout list ends (a close brace is inserted). A close brace is also inserted whenever the syntactic category containing the layout list ends; that is, if an illegal lexeme is encountered at a point where a close brace would be legal, a close brace is inserted. The layout rule matches only those open braces that it has inserted; an explicit open brace must be matched by an explicit close brace.

2. Records

The first Timber extension beyond Haskell is a system for programming with first-class records. Although Haskell already provides some support for records, we have chosen to replace this feature with the present system, partly because the Haskell proposal is a somewhat ad-hoc adaptation of the datatype syntax, and partly because Haskell records do not fit very well with the subtyping extension that is described in Section 3.

The distinguishing feature of Timber records is that the treatment of records and datatypes is perfectly symmetric; that is, there is a close correspondence between record selectors and datatype constructors, between record construction and datatype selection (*i.e.*, pattern-matching over constructors), and between the corresponding forms of type extension, which yield subtypes for records and supertypes for datatypes.

Consequently, we treat *both* record selectors *and* datatype constructors as *global* constants. This is the common choice where datatypes are concerned, but not so for records (see, *e.g.*, references [14] and [8]). Nevertheless, we think that a symmetric treatment has some interesting merits in itself, and that the ability to form hierarchies of record types alleviates most of the problems of having a common scope for all selector names. We also note that overloaded names in Haskell are given very much the same treatment, without creating many problems in practice.

A record type is defined in Timber by a global declaration analogous to the datatype declaration described previously. The following example defines a record type for two-dimensional points, with two selector identifiers of type Float.

```
record Point where x,y :: Float
```

The **record** keyword is also used in the term syntax for record construction. We will generally rely on Haskell's layout rule (see "Layout" on page 7) to avoid cluttering our record expressions with braces, as in the following example.

```
origin = record
           x = 0.0
           y = 0.0
```

Since the selector identifiers x and y are global, there is no need to indicate to which record type a record term belongs; the compiler can deduce that origin is a Point. If one wishes to make this clear to the human reader, one can explicitly write

```
origin:: Point
origin = record
           x = 0.0
           y = 0.0
```

but this is entirely redundant. It is a static error to construct a record term where the set of selector equations is not exhaustive for some record type.

Record selection is performed in the conventional way by means of the *dot*-syntax. (Timber also has an operator **.** used to denote function composition; record selectors

should immediately follow the dot, while the operator **.** must be followed by some amount of white space). Record selection binds more tightly than function and operator application, as the following example indicates.

```
dist p = sqrt (sq p.x + sq p.y) where
    sq i = i * i
```

A record selector can moreover be turned into an ordinary prefix function if needed, by enclosing it between (. and), as in

```
xs = map (.x) some_list_of_points
```

Just as algebraic datatypes may take type arguments, so may record types. The following example shows a record type that captures the signatures of the standard equality operators.†

```
record Eq a where
    eq :: a -> a -> Bool
    ne :: a -> a -> Bool
```

This defines a record type with two selectors named eq and ne. The types of these selectors are both a -> a -> Bool, where the type a is a parameter to Eq.

A record term of type Eq Point is defined below.

```
pdict =  record
             eq = eq
             ne a b = not (eq a b)
         where eq a b = a.x==b.x && a.y==b.y
```

This example also illustrates three minor points about records: (1) record expressions are not recursive, (2) record selectors possess their own namespace (the equation eq = eq above is *not* recursive), and (3) selectors may be implemented as functions if so desired.

†. Strictly speaking, this record type is not legal since its name coincides with that of a predefined Haskell *type class*. Type classes form the basis of the *overloading* system of Haskell, whose ins and outs are beyond the scope of this survey. The name Eq has a deliberate purpose, though — it connects the example to a known Haskell concept, and it indicates the possibility of reducing the number of constructs in Timber by eliminating type class declarations in favour of record types.

3. Subtyping

The subtyping system of Timber is based on *name inequality*. This means that a possible subtype relationship between (say) two record types is determined solely by the names of the involved types, and not by consideration to whether the record types in question might have matching substructure. Name inequality is a generalization of the *name equality* principle used in Haskell for determining whether two types are equal.

The subtype relation between user-supplied types is induced as a consequence of declaring a new type as an extension of a previously defined type. This makes record subtyping in Timber look quite similar to interface extension in Java, as the following type declaration exemplifies:

```
record CPoint < Point where
    color :: Color
```

This syntax both introduces the record type CPoint and declares it to be an extension of the existing type Point. The extension is the addition of the selector color. As a consequence of this definition, CPoint is a subtype of Point, written CPoint < Point. We call CPoint < Point a subtyping rule. The meaning of this definition is that type CPoint possess the selectors x and y in addition to its own selector color.

The structure of CPoint must be observed when constructing CPoint terms, as is done in the following function.

```
addColor :: Point -> CPoint

addColor p = record  x = p.x
                     y = p.y
                     color = Black

cpt = addColor origin
```

Here addColor is defined to be a function that converts Points to CPoints by coloring them black. Notice that leaving out the equation color = Black would make the definition invalid, since the function result would then be a value of type Point instead of CPoint, contradicting the type definition.

Subtyping can also be defined for algebraic datatypes. Consider the following type modelling the colors black and white.

```
data BW = Black | White
```

This type can now be used as the basis for an extended color type:

```
data Color > BW =
        Red | Orange | Yellow | Green | Blue | Violet
```

Since its set of possible values is larger, the new type Color defined here must necessarily be a *supertype* of BW (hence we use the symbol > instead of < when extending a

datatype). The subtype rule introduced by this declaration is accordingly BW < Color, and type Color possess all of the constructors of its base type BW, in addition to those explicitly mentioned for Color. This is analogous to situation for record types formed by extension, where the extended type has all of the destructors (selectors) of its base type.

Timber allows pattern-matching to be incomplete, so there is no datatype counterpart to the static exhaustiveness requirement that exists for record types. However, the set of constructors associated with each datatype still influences the meaning of Timber programs. This is because the type inference algorithm approximates the domain of a pattern-matching construct by the smallest type that contains all of the enumerated constructors. The functions f and g, defined as

```
f Black  = 0
f _      = 1

g Black  = 0
g Red    = 1
g _      = 2
```

illustrate this point. The domain of f is inferred to be BW, while the domain of g is inferred to be Color.

Polymorphic subtype rules

Subtype definitions may be polymorphic. Consider the following example where a record type capturing the standard set of comparison operators is formed by extending the type Eq defined above.

```
record Ord a < Eq a where
    lt, le, ge, gt :: a -> a -> Bool
```

The subtype rule induced by the definition of Ord states that for all types a, a value of type Ord a also supports the operations of Eq a. Ord a must also support the operations lt, le, ge and gt

Polymorphic subtyping works just as well for datatypes. Consider the following example, which provides an alternative definition of the standard Haskell type Either.

```
data Left a = L a

data Right a = R a

data Either a b > Left a, Right b
```

The first declaration, actually defines both a new datatype Left a and a new constructor for values of that type, called L. The declaration of Right is parallel.

The last declaration is an example of type extension with multiple basetypes. Like interface extension in Java, this declaration introduces *two* polymorphic subtype rules; one that says that for all a and b, a value in Left a also belongs to Either a b, and one that says that for all a and b, a value in type Right b also belongs to Either a b. The declara-

tion of Either also shows that a datatype declaration need not declare any new constructors.

Depth subtyping

Subtyping is a reflexive and transitive relation. This, any type is a subtype of itself, and S < T and T < U implies S < U for all types S, T, and U. The fact that type constructors may be parameterized makes subtyping a bit more complicated, though. For example, under what circumstances should we be able to conclude that Eq S is a subtype of Eq T?

Timber incorporates a flexible rule that allows *depth subtyping* within a type constructor application, by taking the *variance* of a type constructor's parameters into account. By variance we mean the rôle that a type variable has in the set of type expressions in its scope—does it occur in a function argument position, in a result position, in both these positions, or perhaps not at all?

In the definition of the record type Eq above

```
record Eq a where
    eq :: a -> a -> Bool
    ne :: a -> a -> Bool
```

all occurrences of the parameter a are in an argument position. For these cases Timber prescribes *contravariant* subtyping, which means that Eq S is a subtype of Eq T only if T is a subtype of S. Thus we have that Eq Point is a subtype of Eq CPoint. This means that an equality test developed for Points can also be applied to CPoints, *e.g.*, it can be used to partition colored points into equivalence classes.

The parameter of the datatype Left, on the other hand, occurs only as a top-level type expression (that is, in a result position). In this case subtyping is *covariant*, which means for example that Left CPoint is a subtype of Left Point. As an example of *invariant* subtyping, consider the record type

```
record Box a where
    in  :: a -> Box a
    out :: a
```

Here the type parameter a plays the rôle of a function argument as well as a result, so both the co- and contravariant rules apply at the same time. The net result is that Box S is a subtype of Box T only if S and T are identical types. There is also the unlikely case where a parameter is not used at all in the definition of a record or datatype:

```
data Contrived a = Unit
```

Clearly a value of type Contrived S also has the type Contrived T for any choice of S and T, thus depth subtyping for this *nonvariant* type constructor can be allowed without any further preconditions. The motivation behind these rules is of course the classical rule for subtyping of function types, which states that S -> T is a subtype of S' -> T' only if S' is a subtype of S, and T is a subtype of T' [7]. Timber naturally supports this rule, as well as covariant subtyping for the built-in aggregate types: lists, tuples, and

arrays. Depth subtyping may be transitively combined with declared subtypes to deduce subtype relationships that are intuitively correct, but perhaps not immediately obvious. Some illustrative examples follow.

Relation:	Interpretation:
Left CPoint < Either Point Int	*If either some kind of point or some integer is expected, a colored point will certainly do.*
Ord Point < Eq CPoint	*If an equality test for colored points is expected, a complete set of comparison operations for arbitrary points definitely meets the goal.*

Restrictions on subtyping

The general rule when defining types by subtyping is that the newly defined subtype or supertype may be any type expression that is not a variable. There are, however, some restrictions, for example, it is illegal to define **record** S a < Bool, because the supertype is not a record type. We will not dwell on the restrictions here; more information can be found in reference [15].

4. Automatic type inference

In a polymorphic language, expressions have several types. A *principal type* is a type sufficiently general for all of the other types to be deducible from it.

In Haskell, the polymorphic function

```
twice f x = f (f x)
```

has the principal type

```
(a -> a) -> a -> a
```

from which every other valid type for twice, *e.g.*, (Point -> Point) -> Point -> Point, can be obtained as a substitution instance.

However, it is well known that polymorphic subtyping systems need types qualified by *subtype constraints* in order to preserve a notion of principal types. To see this, assume that we allow subtyping, and that CPoint < Point. Now twice can also have the type

```
(Point -> CPoint) -> Point -> CPoint
```

which is not an instance of the principal Haskell type. In fact, there can be no simple type for twice that has both (Point -> Point) -> Point -> Point and (Point -> CPoint) -> Point -> CPoint as substitution instances, since the greatest common anti-instance of these types, (a -> b) -> a -> b, is not a valid type for twice.

Thus, to obtain a notion of principality in this case, we must restrict the possible instances of a and b to those types that allow a subtyping step from b to a; that is, we must associate the subtype constraint b < a with the typing of twice. In Timber, subtype

constraints are attached to types using the " =>" symbol†, so the principal type for twice can be written

```
(b < a ) => (a -> b) -> a -> b .
```

This type has two major drawbacks compared to the principal Haskell type: (1) it is syntactically longer than most of its useful instances because of the subtype constraint, and (2) it is no longer unique modulo renaming, since it can be shown that, for example,

```
(b < a, c < a, b < d) => (a -> b) -> c -> d
```

is also a principal type for twice. In this simple example the added complexity that results from these drawbacks is of course manageable, but even just slightly more involved examples soon get out of hand. The problem is that, in effect, *every application node* in the abstract syntax tree can give rise to a new type variable and a new subtype constraint. Known complete inference algorithms tend to illustrate this point very well, and even though algorithms for simplifying the type constraints have been proposed that alleviate the problem to some extent, the general subtype constraint simplification problem is at least NP-hard. It is also an inevitable fact that no conservative simplification strategy can ever give us back the attractive type for twice that we have in Haskell.

For these reasons, Timber relinquishes the goal of complete type inference, and employs a *partial* type inference algorithm that gives up generality to gain consistently readable output. The basic idea is to let functions like twice retain their original Haskell type, and, in the spirit of monomorphic object-oriented languages, infer subtyping steps only when both the inferred and the expected type of an expression are known. This choice can be justified on the grounds that (a -> a) -> a -> a is still likely to be a sufficiently general type for twice in most situations, and that the benefit of consistently readable output from the inference algorithm will arguably outweigh the inconvenience of having to supply a type annotation when this is not the case. We certainly do not want to prohibit exploration of the more elaborate areas of polymorphic subtyping that need constraints, but considering the cost involved, we think that it is reasonable to expect the programmer to supply the type information in these cases.

As an example of where the lack of inferred subtype constraints might seem more unfortunate than in the typing of twice, consider the function

```
min x y = if less x y then x else y
```

which, assuming less is a relation on type Point, will be assigned the type

```
Point -> Point -> Point
```

by Timber's inference algorithm. A more useful choice would probably have been

```
(a < Point) => a -> a -> a
```

---

†. The syntax is inspired by the way that type classes are expressed in Haskell.

but, as we have indicated, such a constrained type can only be attained in Timber by means of an explicit type annotation. On the other hand, note that the *principal* type for min,

```
(a < Point, b < Point, a < c, b < c) => a -> b -> c
```

is still more complicated, and unnecessarily so in most realistic contexts.

An informal characterization of our inference algorithm is that it improves on ordinary polymorphic type inference by allowing subtyping under function application when the types are known, as in

```
addColor cpt
```

In addition, the algorithm computes least upper bounds for instantiation variables when required, so that, *e.g.*, the list

```
[cpt, pt]
```

will receive the type

```
[Point]
```

Greatest lower bounds for function arguments will also be found, resulting in the inferred type

```
CPoint -> (Int,Bool)
```

for the term

```
\ p -> (p.x, p.color == Black)   .
```

Notice, though, that the algorithm assigns constraint-free types to *all* subterms of an expression, hence a compound expression might receive a less general type, even though its principal type has no constraints. One example of this is

```
let twice f x = f (f x) in twice addColor pt
```

which is assigned the type Point, not the principal type CPoint.

Unfortunately, a declarative specification of the set of programs that are amenable to this kind of partial type inference is still an open problem. Completeness relative to a system that lacks constraints is also not a realistic property to strive for, due to the absence of principal types in such a system. However, experience strongly suggests that the algorithm is able to find solutions to most constraint-free typing problems that occur in practice—in fact, an example of where it mistakenly fails has yet to be found in our experience with O'Haskell and Timber programming. Moreover, the algorithm is provably complete with respect to the Haskell type system, and hence possesses another very important property: programs typeable in Haskell retain their inferred types when con-

sidered as Timber programs. Additionally, the algorithm can also be shown to accept all programs typeable in the core type system of Java (see section 5.3 of [15]).

5. Reactive objects

The dynamic behavior of a Timber program is the composition of the behavior of many state-encapsulating, time sensitive *reactive object*s executing concurrently. In this section we will survey this dynamic part of the language as it is seen by the programmer. The number of new concepts introduced here is quite large, so we proceed step by step and ask the reader to be patient while the structure of the language unfolds.

Some of the language features will initially appear to be incompatible with a purely functional language. However, it is in fact the case that all constructs introduced in this section are syntactically transformable into a language consisting of only the Haskell kernel and a set of primitive monadic constants. This "naked" view of Timber will not be pursued here; the interested reader is referred to chapter 6 of Nordlander's Thesis [15].

Objects and methods

Objects are created by executing a **template** construct, which defines the *initial state* of an object together with its *communication interface*. Unlike Smalltalk or Java, there is no notion of class; in this way Timber is similar to classless languages like Emerald [16] and Self [17].

The communication interface can be a value of any type, but it will usually contain one or more *methods*. (It can be useful to put other things in the communications interface too, as we will see in some of the examples.) Methods allow the object to react to *message sends*. We use the term *message send* in its usual sense in object-oriented programming: a message send is directed from one object (the sender) to another object (the receiver); in response, the receiver first selects and then executes a method. Message send is sometimes called method invocation or even method call, but the term message send is preferred because it emphasises that the coupling between the message and the method occurs in the receiving object.

A method takes one of two forms: an asynchronous **action** or a synchronous **request**. An **action** lets the sender continue immediately, and thus introduces concurrency. Consequently, actions have no result. A synchronous **request** causes the sender to wait for the method to complete, but allows a result value to be passed back to the sender.

The body of a method, finally, is a sequence of *commands*, which can basically do three things: update the local state, create new objects, and send messages to other objects. The following template defines a simple counter object:

```
counter =   template
                val := 0
            in record
                    inc =
                        action val := val + 1
                    read =
                        request return val
```

Executing this **template** command creates a new counter object with its own state variable val, and returns an interface through which this new object can be accessed. The interface is a record containing two methods: an asynchronous **action** inc, and a synchronous **request** read. Sending the message inc to this interface will cause the **action** val := val + 1 to be executed, with the effect that the counter object behind the interface will update its state, concurrently with the continued execution of the sender. In contrast, sending the message read will essentially perform a rendezvous with the counter object, and return its current value to the sender.

Procedures and commands

Actions, requests and templates are all expressions that denote commands; such expressions have a type in a *monad* called Cmd. A monad is a structure that allows us to deal with effect-full computations in a mathematically consistent way by formalizing the distinction between *pure* computations (those that simply compute values), and *impure* computations, which may also have effects, such as changing a state or accessing an external device.

Cmd is actually a type constructor: Cmd a denotes the type of commands that may perform *effects* before returning a value of type a. If Counter is a record type defined as

```
record Counter where
    inc :: Cmd ()
    read :: Cmd Int
```

then the type of the example shown above is given by

```
counter :: Cmd Counter
```

This says that counter is a command that, when executed, will return a value of type Counter.

The result returned from the execution of a monadic command may be bound to a variable by means of the *generator* notation. For example

```
do  c <- counter
    ...
```

means that the command counter is executed (*i.e.*, a counter object is created), and c is bound to the value returned—the interface of the new counter object.

Note that a generator expression cannot be the final command in a **do**-construct. There would be no point in using it in such a position, because the bound variable could never be referenced. Executing a command and discarding the result is written without the

left-arrow. For example, the result of invoking an asynchronous method is always the uninteresting value (), so the usual way of incrementing counter c is

```
do  c <- counter
    c.inc
```

This **do**-construct is by itself an expression: it represents a command sequence, that is, an anonymous procedure, like progn in Lisp or like a block in Smalltalk. *When this command sequence is executed,* it executes its component commands in sequence. Thus, the command counter is executed first. This in turn executes the **template** expression shown on page 17, which has the effect of creating a counter object and returning its interface record, which is bound to c. Next, the command c.inc is executed, which sends the message inc to the counter object. The result is ().

The value returned by a procedure is the value returned by its last command, so the type of the above expression is Cmd ().

Since the read method of c is a synchronous **request** that returns an Int, we can write

```
do  c <- counter
    c.inc
    c.read
```

and obtain a procedure with the type Cmd Int.

Just as for other commands, the result of executing the read method can be captured by means of the generator notation:

```
do  c <- counter
    c.inc
    v <- c.read
    return v
```

This procedure is actually equivalent to the previous one. The identifier return denotes the built-in command that, when executed, produces a result value (in this case simply v) without performing any effects. Unlike most imperative languages, however, return is not a branching construct in Timber—a return in the middle of a command sequence means only that a command without effect is executed and its result is discarded. This is pointless, but not incorrect. For example

```
do  ...
    return (v+1)
    return v
```

is simply identical to

```
do  ...
    return v
```

A procedure constructed with **do** can be named just like any other expression:

```
testCounter = do  c <- counter
                  c.inc
                  c.read
```

testCounter is thus the name of a simple procedure that, *when executed*, creates a new counter object and returns its value after it has been incremented once. The counter itself is then simply forgotten, which means that space used in its implementation can be reclaimed by the garbage collector.

A very useful procedure with a predefined name is

```
done = do return ()
```

which plays the rôle of a *null* command in Timber.

Notice the difference between the equals sign = and the generator arrow <-.

- The symbol = denotes definitional equality: it defines the name on the left to be equivalent to the expression on the right, which may be a command (as in this example) or a simple value, such as a factorial function defined on page 5; = can appear at the top level, or in **let** and **where** clauses. Since a definition can never have any effect, the order of execution of a set of definitions made with = is irrelevant, and mutual recursion is allowed.
- In contrast, the generator arrow <- can appear only inside a **do**-construct. The right hand side must be a command, that is, it must have the type Cmd a for some value type a. The effect is to *execute* the command, and to bind the resulting value to the identifier on the left hand side. A sequence of generator bindings is executed in the order written, and the values bound by the earlier bindings can be used in the later expressions, but not vice-versa.

Notice that the definition of counter in “Objects and methods” on page 16 does not by itself create an object. What it does is define counter to be a command. If and when that command is executed, an object will be created; if counter is executed three times, three new counter objects are created. Thus, the command counter is analogous to the expression CounterClass new in Smalltalk or new Counter() in Java: it is the means to make a new object.

Assignable local state

The method-forming constructs **action** and **request** are syntactically similar to procedures, but with different operational behaviors. Whereas calling a procedure means that its commands are executed by the caller, sending a message triggers execution of the commands that make up the corresponding method within the receiving object. Thus, methods have no meaning other than within an object, and the **action** and **request** keywords are accordingly not available outside the **template** syntax.

In actions and requests, as well as in procedures that occur within the scope of an object, two additional forms of commands are available: commands that obtain the values of state variables (for example the command return val in the counter object), and commands that assign new values to these variables (e.g. val := val + 1).

The use of state variables is restricted in order to preserve the purely functional semantics of expression evaluation.

- First, references to state variables may occur only within commands. For example

```
template
    x := 1
in x
```

  is statically illegal, since the state variable x is not visible outside the commands of a method or a procedure (*i.e.*, it can only be used inside a **do**, **action** or **request**).

- Secondly, there are no aliases in Timber, which means that state variables are not first-class values. Thus the procedure declaration

```
someProc r = do r := 1
```

  is illegal even if someProc is applied only to integer state variables, because r is syntactically a parameter, not a state variable. Parameterization over some unknown state can instead be achieved in Timber by turning the parameter candidates into full-fledged objects.

- Thirdly, the scope of a state variable does not extend into nested objects. This makes the following example ill-formed:

```
template
    x := 1
in
    template
        y := 2
    in
        do x := 0
```

- Fourthly, there is a restriction that prevents other local bindings from shadowing a state variable. An expression like the following is thus disallowed:

```
template
    x := 1
in \ x -> ...
```

  While not necessary for preserving the purity of the language, this last restriction has the merit of making the question of assignability a simple lexical matter, as well as emphasizing the special status that state variables enjoy in Timber.

A word about overloading

Sequencing by means of the **do**-construct, and command injection (via return), are not limited to the Cmd monad. Indeed, just as in Haskell, these fundamental operations are *overloaded* and available for any type constructor that is an instance of the *type class* Monad [9, 11]. Type classes and the overloading system will not be covered in this paper, partly because this feature constitutes a virtually orthogonal complement to the subtyping system of Timber, and partly because we do not capitalize on overloading in any essential way. In particular, monadic programming in general will not be a topic of this paper.

Nevertheless, we are about to introduce one more monad that is related to Cmd by means of subtyping. We will therefore take the liberty of reusing return and the **do**-syntax for this new type constructor, even though strictly speaking this means that the overloading system must come into play behind the scenes. The same trick is also employed for the equality operator == in a few places. However, the uses of overloading that occur in this paper are all statically resolvable, so our naive presentation of the matter is intuitively quite correct. We feel that glossing over Haskell's most conspicuous type system feature in this way avoids more confusion than it creates.

The O monad

While all commands are members of the monad Cmd, commands that refer to or assign to the local state of an object belong to a richer monad O s, where s is the type of the local state. Accordingly, O s a is the type of state-sensitive commands that return results of type a. An assignment command always returns (), whereas a state-referencing command can return any type. Any procedure that contains a state-referencing command is itself a state-referencing command, and will therefore have a type O s.

The type of the local state of an object with more than one state variable is a tuple type; there is no information about the names of the state variables encoded in the type of the state. For example, consider the definitions

```
a = template
      x := 1
      f := True
in ...
```

and

```
b = template
      count := 0
      enable := False
in ...
```

The commands a and b, when executed, both generate objects with local states of type (Int,Bool).

Procedures defined within an object are *parametric* in the state on which they operate. The state of the object within which the procedure is eventually executed is, in effect, provided to it as an implicit parameter. There exists no connection at runtime between a value of some O type (a procedure or a method) and the object in which its definition is syntactically nested.

What this means is that, as long as the state types match, a procedure declared within one object can be used as a local procedure within another object. This does not constitute a loophole in Timber's object encapsulation, because the state accessed by that procedure will be the state of the caller. It remains true that the only way in which an object may affect the state of another object is by sending a message to *that object*. However, the ability to export procedures provides a way of sharing code between templates. It is very much like inheritance in class-based languages, which permits one class to re-use the code originally defined in another. In such a language, encapsulation is preserved

because the state on which the re-used code operates that of the currently executing object.

Object Identity

A sometimes controversial issue in the design of object-oriented languages is whether clients should be able to compare two object references for identity. That is, given two identifiers a and b that name objects, can a client ask whether a and b are in fact the same object?

Allowing such an identity test breaches encapsulation, because whether two possibly distinct interfaces are actually implemented by the same object is an implementation decision that may be changed and which should accordingly be hidden. However, failing to provide an efficient identity test can impose an unreasonable burden on the programmer. For a more complete discussion of these issues, see reference [6].

Timber makes the compromise of letting the programmer decide whether or not identity comparison shall be possible. Objects themselves cannot be compared, so encapsulation is preserved. However, Timber provides a special variable self, which is implicitly in scope inside every **template** expression, and which may not be shadowed. All occurrences of self have type Ref s, where s is the type of the current local state. The value of self uniquely identifies a particular object at runtime.

It should be noted that the variable self in Timber has nothing to do with the *interface* of an object (in contrast to, for example, this in C++ and Java). This is a natural consequence of the fact that a Timber object may have multiple interfaces — some objects may even generate new interfaces on demand (recall that an interface is simply a value that contains at least one method).

To facilitate straightforward comparison of arbitrary object reference values, Timber provides the primitive type ObjRef with the built-in subtype rule

```
Ref a < ObjRef   .
```

By means of this rule, all object references can be compared for equality (using the overloaded primitive ==) when considered as values of the supertype ObjRef. Timber moreover provides a predefined record type ObjIdentity, which forms a convenient base from which interface types supporting a notion of object identity can be built.

```
record ObjIdentity where
    self :: ObjRef
```

For example, suppose that we wish to define ICounter as a subtype of the counter type whose objects can be compared for identity.

```
record ICounter < ObjIdentity, Counter

iCounter :: Template ICounter
```

```
iCounter =  template
                c <- counter
            in record
                self = self
                inc = c.inc
                read = c.read
```

Now we can compare the identity of two iCounters:

```
do  c1 <- iCounter
    c2 <- iCounter
    return (c1 == c2)        -- always false
```

Expressions vs. commands

Although commands are first-class values in Timber, there is a sharp distinction between the *execution* of a command, and the *evaluation* of a command considered as a functional value. The following examples illustrate this point.

```
f :: Counter -> (Cmd (), Cmd Int)
f cnt = (cnt.inc, cnt.read)
```

The identifier f defined here is a function, not a procedure: it cannot be *executed*; it can only be applied to arguments of type Counter. The fact that the returned pair has command-valued components does not change the status of f. In particular, the occurrence of sub-expressions cnt.inc and cnt.read in the right-hand side of f does *not* imply that the methods of some counter object are invoked when evaluating applications of f. Extracting the first component of a pair returned by f is also a pure evaluation with no side-effects. However, the result in this case is a command value, which has the specific property of being *executable*.

By placing a command in the body of a procedure, the command becomes subject to execution, *whenever the procedure itself is executed*. Such a procedure is shown below.

```
do  c <- counter
    fst (f c)
```

The second line applies f to a counter object, resulting in a pair. The standard function fst extracts the first element of the pair, which is a command. This command is executed when the procedure (the **do** construct) in which it is defined is executed.

The separation between *evaluation* and *execution* of command values can be made more explicit by introducing a name for the evaluated command. This is achieved by the **let**-command, which is a purely declarative construct: as usual, the equality sign denotes definitional equality.

```
do  c <- counter
    let newCmd = fst (f c)      -- Now newCmd denotes a command
        newCmd                  -- this causes the command to be executed
```

Hence the two preceding examples are actually equivalent, and in each case a counter will be created once and incremented once. The following fragment is yet another equivalent example,

```
do   c <- counter
     let aCmd = fst (f c)      -- Now aCmd and bCmd
         bCmd = fst (f c)      -- both name the same command
     in aCmd
```

whereas the next procedure has a different operational behaviour (here the inc method of c will actually be invoked twice).

```
do   c <- counter
     let newCmd = fst (f c)
     in  newCmd
         newCmd
```

A computation that behaves like f above, but which also has the effect of incrementing the counter it receives as an argument, must be expressed as a procedure.

```
g :: Counter -> Cmd (Cmd (), Cmd Int)
g cnt = do   c.inc
             return (c.inc, c.read)
```

Note that the type system clearly separates the effectfull computation from the pure one: the result type of f is a value, whereas the result type of g is a command.

Likewise, the type system demands that computations that depend on the current state of some object be implemented as procedures. For example,

```
h :: Counter -> Int
h cnt = cnt.read * 10
```

is not type correct, since cnt.read is not an integer — it is a *command* that, *when executed*, returns an integer. If we really want to compute the result of multiplying the counter value by 10 we can write

```
h :: Counter -> Cmd Int
h cnt = do   v <- cnt.read
             return (v * 10)
```

The fact that h calls the read method of the counter is reflected in the return type of h, which is Cmd Int.

Subtyping in the O monad

We have already indicated that the Cmd and O s monads are related by subtyping. This is formally expressed as a built-in subtype rule.

```
Cmd a < O s a
```

This rule can be read as a higher-order relation: “all commands in the monad Cmd are also commands in the monad O s, for any s”.

One way of characterizing the Cmd monad is as a refinement of the O monad that represents those commands that are independent of the current local state. Timber takes this idea even further by providing three more primitive command types, which are related to the Cmd monad via the following built-in subtyping rules.

```
Template a < Cmd a
Action < Cmd ()
Request a < Cmd a
```

The intention here is to provide more precise typings for the **template**, **action**, and **request** constructs. For example, Template a is the type of a **template** that, when executed, constructs an object with an interface of type a. Thus, the type inferred for counter defined on page 17 is actually Template Counter (rather than Cmd Counter), and the types of its two methods are Action and Request Int. The record type Counter can of course be updated to take advantage of this increased precision.

```
record Counter where
    inc :: Action
    read :: Request Int
```

Unlike the refinement step of going from O s to Cmd, which actually makes more programs typeable because of the rank-2 polymorphism, the distinction between Cmd and its subtypes has mostly a documentary value. However, by turning a documentation practice into type declarations, the type system can be relied on to guarantee certain operational properties. For example, a command of type Template a cannot change the state of any existing objects when executed: object instantiation only *adds* objects to the system state. Moreover, commands of type Action or Template a are guaranteed to be deadlock-free, since a synchronous method can never possess any of these types. Note that none of these properties hold for a general command of type Cmd a.

Of the type constructors mentioned here, Cmd, Template, and Request are all covariant in their single argument. This also holds for the type O s a in case of its second argument. However, the O constructor, like all types that support both dereferencing and assignment, must be invariant in its state component. Similarly, Ref is also invariant.

The main template

So far, we have seen how to define functions and procedures, and have emphasized that procedures are executed only when some other procedure calls them. How, then, is the execution of a Timber program started?

A Timber program should have a special template called main. This template is parameterized by an *environment* that gives the program the ability to interact with the rest of the system. The type of the main template must be

```
main :: Environment -> Template Program
```

The definitions of the types Environment and Program depend on the capabilities of the particular system for which the Timber program is written. The section “Reactivity” on page 27 provides more details of these types.

When a Timber program is started, the system will apply main to an environment parameter. The resulting template is then used by the system to create an object which constitutes the system's interface to the running program. This interface is required to contain an action start, which the system executes to initialize the program.

Here is the traditional “Hello World” program in Timber:

```
main env = template
               -- no state
           in record
               start = action
                           env.putStr "Hello World!\n"
```

Concurrency

In general, execution of a Timber program is concurrent: many commands may potentially be active simultaneously. However, each Timber object behaves like a monitor: its methods execute in mutual exclusion, so at most one of its methods can be active at any given time. Since all state is encapsulated in some object, this ensures orderly update of the state.

In the following example, two contending clients send messages to a counter object. Mutual exclusion between method executions in the counter guarantees that there is no danger of simultaneous updates to the counter’s state.

```
proc cnt = template
               -- this object has no state of its own
           in record
                  doIt = action cnt.inc

f env =    do c <- counter
               p <- proc c
               p.doIt
               c.inc
               v <- c.read
               env.putStr (show v)
```

Here proc is function that returns a template (a particular kind of Cmd). The command p <- proc c (inside the **do**) parameterizes proc by the counter object c and *executes* proc c: the result is a new object with a single method called doIt. The message send p.doIt starts execution of this method, which then executes autonomously and asynchronously, because the method is an action.

Methods are not guaranteed to be executed in the order that the corresponding messages are sent. Their execution is instead scheduled subject to timing constraints, which will be discussed in Section 6. However, in the absence of explicit timing constraints, if one message send precedes (in the sense of Lamport’s “happened before” relation) another send to the same object, it is safe to assume that the corresponding methods will be exe-

cuted in the same order. This is also illustrated by the previous example, as we now explain.

In the procedure f the message inc is sent to p before the message read. There are no explicit timing annotations on the message sends c.inc and c.read. Thus, inc will be executed before read, and it is safe to assume that the value v returned from the read request is at least 1. In contrast, the send of doIt to p initiates a concurrent activity, because doIt is an action. Nothing can be said about whether the doIt action of object p will be scheduled to send its inc message before, in between, or after the two messages sent in procedure f.

Reactivity

Objects alternate between indefinitely long phases of inactivity and periods of method execution that must be finite, unless the programmer has explicitly written an infinite loop. Given a sufficiently fast processor, in many applications the method executions may be considered to be instantaneous. When used with very short duration methods, Timber then approximates Berry's *perfectly synchronous* model of computation [3].

The existence of value-returning synchronous methods does not change the fact that method executions are finite, since, assuming that the system is not in deadlock, there are no *other* commands that may block indefinitely, and hence sending a request cannot block indefinitely either. Thus, it is important that the computing environment also adheres to this reactive view, by *not* providing any operations that might block a process indefinitely.

This means, for example, that a Timber program cannot read input from a console using a blocking primitive. Instead, interactive Timber programs install *callback methods* in the computing environment, with the intention that these methods will be invoked whenever the event that they are set to handle occurs. As a consequence, Timber programs do not generally terminate when the start action of the main template returns; instead, they are considered to be alive as long as there is at least one active object or one installed callback method in the system. (Alternatively, the environment may provide a quit method that terminates the whole program).

The overall form of a Timber program is thus *not* a (potentially) infinite main loop. Instead, a Timber program defines a set of objects and binds events in the environment to message sends to those objects. When the events occur, the messages will be sent, and the corresponding methods will be scheduled for execution.

The actual shape of the interface to the computing environment must of course be allowed to vary with the type of application being constructed. The current Timber implementation supports several environment types, including TixEnv, BotEnv, and StdEnv, which model the computing environments offered by a Tk server with graph building extensions [2], a mobile Robot, and the *stdio* fragment of a Unix operating system, respectively.

As an illustration of the use of environments, let us see how a text-based Timber program might work in a minimal Unix-like computing environment:

```
record StdEnvironment where
        putStr :: String -> Action
        quit   :: Action

record StdProgram where
        start  :: Action
        char   :: Char -> Action
        signal :: Int -> Action

main :: StdEnvironment -> Template StdProgram
```

The program can send the message putStr to the environment to output strings. The system will deliver characters and signals to the program by executing the actions char and signal when a new character is typed or a signal is generated.

For an example of a more elaborate environment interface, the reader is referred to the vehicle controller discussed in Section 8.

6. Time

So far, our discussion of Timber has (intentionally) avoided the topic of time. This is conventional in the definition of programming languages; ignoring time has the great advantage of allowing conforming implementations of a language to exist on varied hardware and software platforms. Unfortunately, ignoring time makes a language unsuitable for programming real-time systems, with the result that embedded systems—almost alone in the universe of modern software—are frequently programmed in assembly language or in a way that must escape from the programming language and appeal to the primitives of a real-time operating system for all critical operations.

With Timber we attempt to find some middle ground by allowing the programmer to place *bounds* on the execution time of actions, while allowing the implementation the freedom to schedule the actions within those bounds. We use the notion of *deadline*—the latest time before which an action must complete, and *baseline*—the earliest time after which the action may commence. We call the closed interval bounded by a baseline and a deadline a *timeline*; while an action is executing the current time will normally be within the timeline for that action. It is possible to read the current time directly, but since this will vary from one execution to the next, the timeline is in practice more useful.

Specifying Time

Timber has two built-in datatypes for time: TimeInstant and TimeDuration. TimeInstant refers to a calendar date and time; TimeDuration to the interval between two TimeInstants†. Neither the precision nor the accuracy of the clock against which

†. The names *TimeDuration* and *TimeInstant* are taken from ISO 8601[10] and the XML Schema Specifications for DataTypes [5]

times are measured are dictated by the Timber language. This means that implementations are free to provide as coarse or as fine a notion of time as their applications require or their platforms permit. However, Timber does require that time is monotonic with respect to the Lamport Logical Clock [13]. That is, if an action a "happened before" an action b, then the current time observable in a must not be later than the current time observable in b. Note that, because of the finite granularity of the clock, the two times may be equal.

The datatype TimeInterval is used to represent the interval between and including two TimeInstants. The operators until, from and ending can be used to construct TimeIntervals:

```
until   :: TimeInstant    -> TimeInstant    -> TimeInterval
from    :: TimeDuration   -> TimeInstant    -> TimeInterval
lasting :: TimeInstant    -> TimeDuration -> TimeInterval
ending  :: TimeDuration   -> TimeInstant    -> TimeInterval
```

In the following, tod1 is a TimeInstant, and hours is a TimeDuration. tod2 is defined to denote a timeInstant that is one hour later than tod1:

```
tod2 = tod1 + (1 * hours)
```

The following definitions all specify the same TimeInterval, namely, the interval between tod1 and tod2.

```
i1 = tod1 `until` tod2
i2 = (1 * hours) `from` tod1
i3 = tod1 `lasting` (1 * hours)
i4 = (1 * hours) `ending` tod2
```

The operators baseline, deadline and duration can be used to examine a TimeInterval:

```
baseline :: TimeInterval   -> TimeInstant
deadline :: TimeInterval   -> TimeInstant
duration :: TimeInterval   -> TimeDuration
```

In the scope of the above let expression, the following are true:

```
baseline i1 == tod1
deadline i1 == tod2
duration i1 == (1 * hours)
```

Timelines for Actions

Every method execution in a Timber program has an associated timeline. Normally, this timeline is the same as the timeline of the method that initiated the execution; indeed, this is always the case for requests. However, for actions, it is possible to specify a different timeline, as we will see shortly.

Actions invoked by the environment are also assigned timelines. For example, the timeline for the start action is determined by the operating system command that initiates it. Normally it has a baseline representing the TimeInstant at which the program is started,

and a deadline specifying when initialization must be completed. To give another example: when the environment receives an interrupt from a sensor, it sends a message to some Timber object that initiates an action. The timeline for this action might extend from the instant that the interrupt arrives until the instant when the sensor readings are no longer guaranteed to be available in the device register.

Specifying the Timeline

If a Timber action with timeline τ sends a message initiating an action *A*, then the default timeline for *A* is also τ. The baseline for *A* can be specified to be something other than τ.baseline than by means of the construct **after** b *A*, where b is a TimeDuration. This gives the action *A* a baseline of b + τ.baseline; *A*'s deadline is τ.deadline. Similarly, the deadline for *A* can be specified by means of the construct **before** d *A*, where d is a TimeDuration; this initiates the action *A* with a deadline of d + τ.baseline. In this case *A*'s baseline is τ.baseline.

The **before** and **after** constructs give the programmer an explicit way of specifying which aspects of a reaction are time-critical. If an action *A* sends a message that initiates an action *B* in some other object, the deadline for *B* will by default be the same as that of *A* itself. However, by using the **before** command, the deadline for *B* can be changed to be later than the deadline for *A*.

Whether it is appropriate to change the deadline in this way depends entirely on the application. For example, *A* may be a time-critical reaction to a real-time event, but *B* may be a housekeeping operation that can be deferred indefinitely; in this case, *B* may be given a very much more generous (even infinite) deadline. In contrast, if completion of *B* is part of the required response to the external event, then it may be necessary to give *B* the same deadline as *A*.

By using a recursive message send that specifies a new baseline, it is possible to express *periodic* scheduling. For example, the following controller schedules itself with a period of 0.1 seconds:

```
controller = action
                do_periodic_stuff
                after (0.1 * seconds) controller
```

It is important to note that the n^{th} execution of this action will have terminated before the $(n+1)^{th}$ execution starts.

Execution Model

The model of concurrent execution used by Timber is based on the idea of the Chemical Abstract Machine [4]. The state of an executing program is envisioned as a "soup" of molecules. Sometimes these molecules react together, becoming absorbed and producing new molecules as a result.

There are two kinds of molecules in the Timber "soup": *objects* and *messages*.

Objects. Objects are always *named*. The names bear no relationship to any identifier that might be used to reference an object in the Timber program. Instead, a name should be thought of as a unique identifier that distinguishes an object from all others.

Objects can either be *active* or *inactive*. An active object is denoted o:Obj⟨C, τ, s⟩. Such an object, named o, is executing the command sequence C in response to a message sent from the object named s, using the timeline τ. An inactive object is denoted o:Obj⟨⟩.

Messages. Messages are denoted Msg⟨o, C, τ, s⟩, which amounts to a message targeted at object o, containing the command sequence C, to be executed with the timeline τ, on behalf of the invoking object s. If the message corresponds to an invoked action, we will use the special identifier _ for the invoking object.

Object creation. When an object is created by executing a template command, a new object o:Obj⟨⟩ is created, using a fresh name o. The state variables of o is initialized as described by the template, and sub-objects are recursively created. All actions and requests in the template are also *associated* with the name o, so that messages can be sent to the correct target. The interface (containing the associated actions and requests), is returned.

Action message send. When a (asynchronous) action message is sent, a new message of the form Msg⟨o, C, τ, _⟩ is created, where o is the target object associated with the action, C is the command sequence in the action, and τ is the timeline specified for the action (see "Specifying the Timeline" on page 30). The identity of the sender is irrelevant in this case, and so is denoted by _.

Request message send. When a (synchronous) request message is sent, a new message of the form Msg⟨o, C, τ, s⟩ is created, where o is the target object associated with the request, C is the command sequence in the action, and τ is the timeline of the invoking method. The invoking method is blocked, awaiting a reply from o.

Dispatching of a message. If an idle object o:Obj⟨⟩ and a message Msg⟨o, C, τ, s⟩ that targets o both exist at the same time, then the message can be *dispatched*. This means that both o and the message are consumed and are replaced by the active object o:Obj⟨C, τ, s⟩. Note that this dispatch is constrained by the *scheduling rules* outlined in the next section.

Completing an action. When an active object has finished executing an action command sequence, it is transformed into the idle object.

Completing a request. When an active object has finished executing a request command sequence, it is on the form o:Obj⟨**return** e, τ, s⟩. It will be transformed into the idle object o:Obj⟨⟩, and the object s that originally sent the request message is unblocked. The return value of the message send is the value of e.

Scheduling

In the Timber execution model, scheduling reduces to the problem of choosing which message to dispatch next. The exact scheduling algorithm is not a part of the Timber language specification. Instead, we envisage the scheduler as a "plug in component": different schedulers may be chosen to meet the needs of different applications.

However, any scheduler *must* preserve the following properties:

1. No message may be dispatched before its baseline.

2. If two messages to the same object o, A = Msg⟨o, m, τ_1, _ ⟩ and B = Msg⟨o, n, τ_2, _ ⟩ are both eligible for dispatch, and A was sent *before* B, in the sense of Lamport's "happened before" relation, then B can only be dispatched before A if
   - τ_2.baseline < τ_1.baseline, or
   - τ_2.baseline = τ_1.baseline **and** τ_2.deadline < τ_1.deadline.

The second property guarantees that the order is preserved in a sequence of message sends from one object to another, provided that all the messages have the same timeline. However, if a programmer explicitly gives a later message an earlier baseline or an earlier deadline, then the later message may be dispatched before the earlier one.

Example of Reduction Semantics

Recall our definition of the counter template:

```
counter =   template
              val := 0
            in record
              inc    = action val := val + 1
              read   = request return val
```

Suppose we have an active object o:Obj⟨C, τ, _ ⟩, where is C is the following command sequence:

```
c <- counter
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v)
```

Here is how the system can evolve:

```
o:Obj⟨c <- counter
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v), τ, _ ⟩
```

Unfold definition of counter, create new object, with fresh name o1. Return interface with methods associated with o1

```
o:Obj⟨c <- return (record    inc    = action⟨o1⟩ val := val + 1
                             read   = request⟨o1⟩ return val)
     c.inc
     c.inc
     v <- c.read
     env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Bind c to returned expression

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      c.inc
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Evaluate c.inc

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      action⟨o1⟩ val := val + 1
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]
```

Invoke the first action, creating a new message

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 0]

Msg⟨o1,  val := val + 1,   τ,   _ ⟩
```

Dispatch the message (this is just one of many possible schedules)

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
       c.inc
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]
```

Invoke the second action (this is just one of many possible schedules)

```
o:Obj⟨let c = record   inc    = action⟨o1⟩ val := val + 1
                       read   = request⟨o1⟩ return val)
      v <- c.read
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1,  val := val + 1,  τ,   _ ⟩
```

Evaluate c.read (this is just one of many possible schedules), garbage collect c

o:Obj⟨v <- **request**⟨o1⟩ return val
env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1, val := val + 1, τ, _ ⟩

Invoke request (this is just one of many possible schedules)

o:Obj⟨v <- ⟨blocked⟩
env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 0]

Msg⟨o1, val := val + 1, τ, _ ⟩

Msg⟨o1, return val, τ, o⟩

Execute assignment, complete action

o:Obj⟨v <- ⟨blocked⟩
env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 1]

Msg⟨o1, val := val + 1, τ, _ ⟩

Msg⟨o1, return val, τ, o⟩

Dispatch message (scheduling requirements state that this is the only possible message to dispatch for o1)

o:Obj⟨v <- ⟨blocked⟩
env.putStr (show v), τ, _ ⟩

o1:Obj⟨val := val + 1, τ, _ ⟩ [val := 1]

Msg⟨o1, return val, τ, o⟩

Execute assignment, complete action

o:Obj⟨v <- ⟨blocked⟩
env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 2]

Msg⟨o1, return val, τ, o⟩

Dispatch message

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨return val, τ, o⟩ [val := 2]
```

Evaluate local state variable

```
o:Obj⟨v <- ⟨blocked⟩
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨return 2, τ, o⟩ [val := 2]
```

Complete request, unblock invoking object

```
o:Obj⟨v <- return 2
      env.putStr (show v), τ, _ ⟩

o1:Obj⟨⟩ [val := 2]
```

Bind v to returned expression, garbage collect o1

```
o:Obj⟨   let v = 2
         env.putStr (show v), τ, _ ⟩
```

Evaluate expression, garbage collect v

```
o:Obj⟨env.putStr "2", τ, _ ⟩
```

7. Additional Features

Timber also provides a number of minor, mostly syntactic extensions to the Haskell base, which we will briefly review in this section.

Extended do-syntax

The **do** -syntax of Haskell already contains an example of an expression construct lifted to a corresponding role as a command: the **let**-command, illustrated in “Expressions vs. commands” on page 23. Timber defines commands corresponding to the **if**- and **case**-expressions as well, using the following syntax.

```
do   if e then
        cmds
     else
        cmds
     if e then
        cmds
     case e of
        p1 -> cmds
        p2 -> cmds
```

In addition, Timber provides syntactic support for recursive generator bindings, and iteration.

```
do   fix x <- cmd y
         y <- cmd x
     forall i <- e do
         cmds
     while e do
         cmds
```

Array updates

To simplify programming with the primitive Array type, Timber supports a special array-update syntax for arrays declared as state variables. Assuming a is such an array, an update to a at index i with expression e can be done as follows. (The array indexing operator in Haskell is !)

```
a!i := e
```

Semantically, this form of assignment is equivalent to

```
a := a // [(i,e)]
```

where // is Haskell's pure array update operator. But apart from being intuitively simpler, the former syntax has the merit of making it clear that normal use of an encapsulated array is likely to be single-threaded, *i.e.*, implementable by destructive update. The rare cases where a is used for a purpose other than indexing become easily identifiable, and hence conservative of the array can be reserved for these occasions. Ordinary updates to a can be performed in place, which is also exactly what the array-update syntax above suggests.

Record stuffing

Record expressions may optionally be terminated by a type constructor name, as in the following examples:

```
record ..S

record a = exp; b = exp; ..S
```

These expressions utilize *record stuffing*, a syntactic device for completing record definitions with equations that just map a selector name to an identical variable already in scope. The missing selectors in such an expression are determined by the appended type constructor S, which must stand for a record type, on condition that corresponding variables are defined in the enclosing scope. So if S is a (possibly parameterized) record type with selectors a, b, and c, the two record values above are actually

```
record a = a; b = b; c = c
```

and

```
record a = exp; b = exp; c = c
```

where c, and in the first case even a and b, must already be bound. Record stuffing is most useful in conjunction with **let**-expressions, as we will see in the examples.

8. An Autonomous Vehicle Controller

We present here a complete Timber program. The example is idealized for brevity, but illustrates Timber's reactive style of programming and many of the features of the language. This example also shows the separation between the calculations performed by a program and the interactions in which it is involved. Since it is an implementation of an interrupt-driven system with parallel processes that also performs significant computation, it captures many of the characteristics of an embedded system.

The environment that this program assumes is as follows:

```
record Register where
    load   :: Cmd Int
    store  :: Int -> Cmd ()

record EmbeddedEnv where
    register_at   :: Int -> Template Register
    reset         :: Action

record EmbeddedProgram where
    start      :: Action
    interrupts :: [(Int,Action)]
```

Here is the controller program itself:

```
module AGV where

type Angle     = Float
type Speed     = (Angle,Float)
type Pos       = (Float,Float)

calcpos  :: [Angle] -> [Pos] -> Pos
regulate :: Pos -> Pos -> Speed -> Speed
room     :: [Pos]

calcpos     = undefined
regulate    = undefined
room        = undefined

-----------------------------------------------------------------

record Driver where
    new_scan    :: [Angle] -> Action
    new_path    :: [Pos] -> Action
```

```
driver :: Servo -> Template Driver
driver servo =
    template
        speed := (0.0,0.0)
        path  := repeat (0.0,0.0)
    in record
        new_scan angles = action
            let is_pos                = calcpos angles room
                should_pos:path'      = path
                speed'                = regulate is_pos should_pos speed
            speed := speed'
            path  := path'
            servo.set_speed speed'
        new_path p = action
            path := p

-----------------------------------------------------------------------

record Scanner where
    detect      :: Action
    zero_cross  :: Action

tick_period  = 100*milliseconds
reg_change= 10*microseconds

scanner :: Register -> Driver -> Template Scanner
scanner angle_reg driver =
    template
        angles := [ ]
    in record
        detect = beforereg_change action
            a <- angle_reg.load
            angles := 2*pi*(fromIntegral a)/4000 : angles
    zero_cross = action
            before tick_period driver.new_scan angles
            angles := []

-----------------------------------------------------------------------

record Servo where
    set_speed  :: Speed -> Action

servo :: Register -> Register -> Template Servo
servo  = undefined

-----------------------------------------------------------------------

record Radio where
    incoming :: Action

radio :: Register -> Driver -> Template Radio
radio  = undefined
```

```
-----------------------------------------------------------------

main :: EmbeddedEnv -> Template EmbeddedProgram
main env =
    template
        thrust_reg    <- env.register_at 0xFFFF0001
        steer_reg     <- env.register_at 0xFFFF0002
        angle_reg     <- env.register_at 0xFFFF0003
        radio_reg     <- env.register_at 0xFFFF0004

        serv          <- servo thrust_reg steer_reg
        driv          <- driver serv
        scan          <- scanner angle_reg driv
        comm          <- radio radio_reg driv
    in record
        start = actiondone
        interrupts =  [
                          (0x80, scan.detect),
                          (0x81, scan.zero_cross),
                          (0x82, comm.incoming)
                      ]
```

Appendix: A Context-Free Grammar for Timber

Module Header

```
module   : 'module' CONID 'where' body

body     : '{' topdecls '}'
         | topdecls                    -- using layout
```

Top-level declarations

```
topdecls : topdecls ';' topdecl
         | topdecl

topdecl  : 'type' CONID tyvars '=' type
         | 'data' CONID tyvars optsubs optcs
         | 'record' CONID tyvars optsups optbs
         | 'class' CONID tyvars optsups optbs
         | 'instance' qtype optbs
         | bind

tyvars   : tyvars VARID
         | {- empty -}

optsups  : '<' types
         | '<' type
         | {- empty -}

optsubs  : '>' types
         | '>' type
         | {- empty -}
```

Datatype declarations

```
optcs     : '=' constrs
          | {- empty -}

constrs   : constrs '|' qconstr
          | qconstr

qconstr   : context '=>' constr
          | constr

constr    : constr atype
          | CONID
```

Bindings

```
optbs     : 'where' bindlist
          | {- empty -}

bindlist  : '{' binds '}'
          | binds                    -- using layout
          | '..' CONID               -- only in a record expression

binds     : binds ';' bind
          | bind

bind      : vars '::' qtype
          | pat rhs                  -- unless inside a record declaration

vars      : vars ',' var
          | var

rhs       : '=' exp
          | gdrhss
          | rhs 'where' bindlist

gdrhss    : gdrhss gdrhs
          | gdrhs

gdrhs     : '|' quals '=' exp
```

Qualified types

```
qtype     : context '=>' type
          | type

context   : '(' preds ')'
          | pred

preds     : preds ',' pred
          | pred

pred      : classpred
          | type '<' type

classpred : classpred atype
          | CONID
```

Types

```
type      : btype '->' type
          | btype
```

```
btype    : btype atype
         | atype

atype    : CONID
         | VARID
         | '[' ']'
         | '(' '->' ')'
         | '(' commas ')'
         | '(' ')'
         | '(' type ')'
         | '(' types ')'
         | '[' type ']'

types    : types ',' type
         | type  ',' type

commas   : commas ','
         | ','
```

Expressions

```
exp      : '\' apats '->' exp
         | 'let' bindlist 'in' exp
         | 'if' exp 'then' exp 'else' exp
         | 'case' exp 'of' altlist
         | 'record' bindlist
         | 'do' stmtlist
         | 'action' stmtlist
         | 'request' stmtlist
         | 'template' stmtlist 'in' exp
         | 'template' 'in' exp
         | 'after' aexp exp
         | 'before' aexp exp
         | exp '::' qtype
         | infixexp

infixexp : infixexp op infixexp
         | '-' fexp
         | fexp

fexp     : fexp aexp
         | aexp

aexp     : aexp SELID
         | bexp

bexp     : var
         | 'self'
         | con
         | lit
         | '(' ')'
         | '(' exp ')'
         | '(' exps ')'
         | '[' list ']'
         | '(' infixexp op ')'
```

```
                | '(' op infixexp ')'
                | '(' commas ')'
lit             : INT
                | RATIONAL
                | CHAR
                | STRING
```

List expressions

```
list            : {- empty -}
                | exp
                | exps
                | exp '|' quals
exps            : exps ',' exp
                | exp ',' exp
quals           : quals ',' qual
                | qual
qual            : pat '<-' exp
                | exp
                | 'let' bindlist
```

Case alternatives

```
altlist         : '{' alts '}'
                | alts                          -- using layout
alts            : alts ';' alt
                | alt
alt             : pat rhs1
rhs1            : '->' exp
                | gdrhss1
                | rhs1 'where' bindlist
gdrhss1         : gdrhss1 gdrhs1
                | gdrhs1
gdrhs1          : '|' quals '->' exp
```

Statement sequences

```
stmtlist        : '{' stmts '}'
                | stmts                         -- using layout
stmts           : stmts ';' stmt
                | stmt
stmt            : pat '<-' exp
                | exp
                | pat ':=' exp
                | 'let' bindlist
                | 'if' exp 'then' stmtlist 'else' stmtlist
                | 'if' exp 'then' stmtlist
                | 'case' exp 'of' altlist2
                | 'forall' quals 'do' stmtlist
```

```
                | 'while' exp 'do' stmtlist
                | 'fix' stmtlist
altlist2        : '{' alts2 '}'
                | alts2                                  -- using layout
alts2           : alts2 ';' alt2
                | alt2
alt2            : pat rhs2
rhs2            : '->' stmtlist
                | gdrhss2
                | rhs2 'where' bindlist
gdrhss2         : gdrhss2 gdrhs2
                | gdrhs2
gdrhs2          : '|' quals '->' stmtlist
```

Patterns

```
pat             : pat op pat
                | apats
apats           : apats apat
                | apat
apat            : '_'
                | var
                | con
                | lit
                | '-' INT
                | '-' RATIONAL
                | '(' ')'
                | '(' pat ')'
                | '(' pats ')'
                | '[' pats ']'
                | '(' commas ')'
pats            : pats ',' pat
                | pat ',' pat
```

Variables, Constructors and Operators

```
var             : VARID
                | '(' VARSYM ')'
con             : CONID
                | '(' CONSYM ')'
varop           : VARSYM
                | '`' VARID '`'
conop           : CONSYM
                | '`' CONID '`'
op              : varop
                | conop
```

Terminal symbols

Rather than providing full definitions for the terminals, we illustrate them by example.

Variable Identifiers

VARID: abc | aBC | ab_c | abc1 | ...

Constructor Identifiers.

CONID: Abc | ABC | Ab_c | Abc1 | ...

Selector Identifiers:

SELID: .abc | .aBC | .ab_c | .abc1 | ...

Variable Symbols

VARSYM: + | < | <= | ...

Constructor Symbols

CONSYM: : | :+ | :< | :<= | ...

Integers

INT: 0 | 123 | 0x123ABC | ...

Rational Numbers

RATIONAL: 0.12 | 0.12E4 | 0.12E-4 | ...

Character Constants

CHAR: 'a' | 'X' | '\n' | ...

String Constants

STRING: "abc" | "abc\n" | ...

References

1. Haskell—A Purely Functional Language. Web site, http://www.haskell.org/
2. TiX: Tk interface eXtension. Web site, http://tix.sourceforge.net/
3. Gérard Berry, *The Foundations of Esterel*, in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, G. Plotkin, C. Stirling, and M. Tofte, Editors. 1998, MIT Press.
4. Gerard Berry and Gerard Boudol. *The Chemical Abstract Machine*. In *Seventeenth annual ACM symposium on Principles of programming languages*, 1990, San Francisco, CA, USA: ACM Press, .
5. Paul V. Biron and Ashok Malhotra, XML Schema Part 2: Datatypes. Stable W3C Recommendation 02 May 2001, World Wide Web Consortium (W3C), 2001. http://www.w3.org/TR/xmlschema-2/
6. Andrew P. Black. *Object Identity.* In *Proceedings 3rd International Workshop on Object Orientation in Operating Systems*, 1993, Asheville, NC: IEEE Computer Society Press, .
7. Luca Cardelli and Peter Wegner, *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 1985. **17**(4): pp 471-522.
8. Benedict R. Gaster. *Polymorphic Extensible Records for Haskell*. In *Haskell Workshop*, 1997, Amsterdam, The Netherlands.

9. Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones and Philip Wadler. *Type classes in Haskell*. In *5th European Symposium on Programming*, 1994, Edinburgh, Scotland: Springer Verlag, Lecture Notes in Computer Science vol. 788.
10. ISO, Representations of dates and times. 1988-06-15, International Organization for Standardization, 1988.
11. Mark P. Jones. *A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorphism*. In *Functional Programming and Computer Architecture*, 1993, Copenhagen, Denmark: ACM Press, .
12. Simon Peyton Jones, John Hughes, Lennart Augustsson, *et al.*, Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language. , , 1999. http://www.haskell.org
13. Leslie Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System.* Communications of the ACM, 1978. **21**(7): pp 558-565.
14. R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*. 1990, Cambridge, MA: MIT Press.
15. Johan Nordlander. Reactive Objects and Functional Programming [Ph.D. Dissertation]. Chalmers University of Technology, Göteborg, Sweden:1999.
16. R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson and E. Jul, *Emerald: A General Purpose Programming Language.* Software—Practice & Experience, 1991. **21**(1): pp 91-118.
17. David Ungar and Randall B. Smith. *Self: The Power of Simplicity.* In *OOPSLA'87*, 1987.

APPENDIX C

Reactive Objects. Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black, Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002), Arlington, VA, 2002.

Reactive Objects

Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburtz, and Andrew Black*
OGI School of Science & Engineering at OHSU, 20000 NW Walker Road, Beaverton, OR 97006.

Abstract

Object-oriented, concurrent, and event-based programming models provide a natural framework in which to express the behavior of distributed and embedded software systems. However, contemporary programming languages still base their I/O primitives on a model in which the environment is assumed to be centrally controlled and synchronous, and interactions with the environment carried out through blocking subroutine calls. The gap between this view and the natural asynchrony of the real world has made event-based programming a complex and error-prone activity, despite recent focus on event-based frameworks and middleware.

In this paper we present a consistent model of event-based concurrency, centered around the notion of reactive objects*. This model relieves the object-oriented paradigm from the idea of transparent blocking, and naturally enforces reactivity and state consistency. We illustrate our point by a program example that offers substantial improvements in size and simplicity over a corresponding Java-based solution.*

1 Background

In the traditional view of programming, the program is assumed to be the master of its environment, and interaction with the environment is accordingly expressed in terms of the subroutine abstraction. This programming model dates back to the early age of batch-oriented computing, when programmers saw a need to abstract away from low-level details of peripheral devices—such as card-readers and line-printers—that were used at the time. The key idea here is that blocking of execution is made *transparent*; that is, the programmer is supposed not to be interested in knowing whether a subroutine obtains its result by some internal computation, or by means of synchronization with an external device.

Despite many advances in language design, this simple, traditional view of I/O still prevails in contemporary object-oriented languages. But modern software executes in much more complex environments, with interactive point-and-click graphics, ubiquitous networks, multiple threads of activity with inter-thread communication and sharing, and so on. Embedded systems fit this description perhaps even better, with their rich variety of asynchronous input sources, and often clock-driven concurrent processes. In such environments, the utility of the traditional, batch-oriented view of interaction deteriorates rapidly.

The central problem is that, if external input is obtained as the results of certain subroutine calls, a program must make a choice as to what subroutine to call, and hence make a premature commitment to which event it will accept next. But the order of external events is seldom under program control, so a naive adherence to the batch-oriented I/O model quickly leads to programs in which events are either missed, or else randomly reordered in time.

The common pattern of an *event loop* in sequential object-oriented software is an attempt to reduce the rigidity of the traditional I/O model. However, an event-loop solution requires that all events of interest are already encoded externally and posted to a common queue, which may not always be the case. For example, the Java Abstract Window Toolkit (AWT) uses a common event queue for mouse-clicks, key-presses, and other GUI-related events [6]. Network sockets, on the other hand, are not handled by the AWT. So a program that needs to simultaneously monitor GUI events as well as network packets must still battle the limitations of the traditional I/O model: how to wait for multiple asynchronous events – all modelled as the results of distinct method calls to the environment – at the same time?

A standard approach in scenarios like this is to allocate a unique *thread of execution* for each potentially blocking operation, writing the code for each thread as if only one future event mattered. On the surface, such a strategy might appear to simplify the design task, because each thread now fits the traditional, batch-oriented I/O model quite well. However, in any non-trivial application, where the specified reactions are not completely independent, the original problem of coordinating inputs from multiple, asynchronous sources has now moved to another part of the program: namely, the thread in which the results of the simple blocking threads must be coordinated. To solve this problem one must of course face all the well-known problems of concurrent thread programming — assuring thread safety and state consistency, while also

*email: {nordland,mpj,magnus,dick,black}@cse.ogi.edu

ensuring liveness and avoiding deadlocks.

Notice, though, that in the AWT-plus-sockets scenario sketched above there is nothing that actually suggests concurrent *execution*; what the extra threads achieve is really the ability to perform concurrent *blocking*. While concurrency is an important tool in real-time programming, being forced to use it just to circumvent an inappropriate I/O model is not satisfactory. Also note that an abstract object model of the example would probably consist of just one simple object, equipped with methods corresponding to the events it is required to handle: mouse clicks, key presses, and packet arrivals. A central argument of this paper is that the heavy encodings needed to turn even simple event-driven models into working code is most unfortunate, and is an important factor behind the perceived complexity of concurrent object-oriented programming.

The current practice of using threads to circumvent the traditional I/O model is extremely fragile, in the sense that an accidental call to a blocking operation in the middle of an event-handler will immediately destroy the responsiveness of that thread. That is, transparent blocking makes responsiveness a delicate property that can only be upheld by careful programming, requiring knowledge not only of the complex APIs that encode events and threads, but also of all operations which potentially might block, and which thus must be avoided. As an illustration, consider the following quote from the documentation for Java's New I/O library [9]:

> That a selection key indicates that its channel is ready for some operation is a hint, but not a guarantee, that such an operation can be performed by a thread without causing the thread to block. It is imperative that code that performs multiplexed I/O be written so as to ignore these hints when they prove to be incorrect.

Clearly some special skill or rigorous discipline is required to navigate safely through such dangerous waters. It is especially noteworthy that this comment concerns a new library with pretensions to make event-driven Java programming significantly easier than before.

The idea of transparent blocking takes its most sophisticated form in the remote-procedure-call paradigm, or remote-method-invocation (RMI) as it is called using Java terminology. Again, hiding the intricacies of synchronization with a remote machine under a familiar subroutine-like interface seems attractive at first, because it makes the code of distributed programs look quite similar to code written for use in a strictly local context.

However, the similarity is deceptive. A defining aspect of a distributed system is usually that it is subject to *partial failure*; that is, programs are expected to continue running even if a remote server is down, broken, or otherwise unaccessible. Contrast this to failures directly affecting the local node: here the failure of one component is equivalent to the whole node going down. So in order to hide distribution, the RMI paradigm must also hide the possibility of partial failures. What this means is that failure of any remote machine in an RMI setup is equivalent to a total system failure. In practice, RMI-based programs can only regain some form of robustness by protecting the remote invocations by timeouts and exception handlers. However, this of course also makes the RMI paradigm considerably less convenient (and distribution less transparent) [8].

2 Reactive objects

The conclusion we have drawn from the problems associated with transparent blocking and batch-oriented I/O is that significantly more robust event-based software can be obtained by abandoning indefinite blocking altogether, and letting an event-driven design permeate the whole programming model. Thread packages, design patterns, and various middleware layers can only do so much to alleviate the programmer from a fundamentally computer-centric view, and they cannot help at all with enforcing responsiveness as long as every method call has a potential of blocking.

Our alternative programming model takes as its starting point the intuition behind the classical object-oriented paradigm: objects are autonomous, objects maintain a state, objects have methods, methods execute in response to messages. The main step towards a reactive variant of this model is to relieve the classical model from any ties to the traditional way of viewing I/O. In its place, a more orthodox object-oriented scheme of interaction can be devised:

- *input*—the environment calls a method of a program object
- *output*—the program calls a method of an environment object

Method calls in these categories do not just carry data, they can also be seen as representing the actual input and output *events* themselves. Notice in particular the asymmetry between input and output that results from this scheme: output is a concrete *act* of the program, while input is modelled as a passive capability to *react*. In other words, objects have full control over the output events, but leave the input events to be scheduled by the environment.

The autonomy and integrity of objects is essential to this view, though. Just as it is usually beneficial to view real-world objects as having a certain level of atomicity of operation, so is it essential to keep software objects from becoming invaded by multiple method invocations at the same time.

On the other hand, the concurrent operation of distinct objects is a natural aspect of the real world, and we wish our

reactive objects to support the same intuition. We therefore take it as a semantic foundation of our model that

> *every object is an autonomous unit of execution that is either executing the sequential code of exactly one method, or passively maintaining its state.*

The combination of inter-object concurrency with internal sequential execution effectively makes a reactive object a union of the well-known concepts of an encapsulated state and a critical region.

Because objects are autonomous execution units, it makes sense to distinguish between *asynchronous* and *synchronous* method invocations. In the former case, the sender of a message continues execution in parallel with the receiving object, whereas in the latter case, the sender and receiver perform a rendezvous. Of course, only synchronous methods provide an opportunity to directly return a result from the receiver to the sender of a message.

At a first glance, synchronous methods seem to provide a way of reintroducing traditional input methods like `getc` in the model. We do however make an important restriction that will prohibit such use:

> *no methods—in environments or in programs—must block execution indefinitely.*

This restriction rules out language constructs like selective method filtering, as well as environments that provide naive interfaces to blocking system calls. All that a synchronous method call can do is to compute a reply based on the current state of the receiver, possibly after performing some side-effects. The serialization of all method executions of the receiving object does not change the fact that a synchronous method is essentially an ordinary subroutine, since, by an argument of transitivity, if the receiver is not ready to immediately execute a synchronous call, it must be busy servicing one of the non-blocking calls that stand in the way.

Reactive objects thus alternate between phases of passive inactivity and temporary outbursts of method execution. In contrast to so called active objects, a reactive object does not have a continuous thread of execution; all executable code of an object is defined in terms of its methods. A method of a reactive object is furthermore guaranteed to terminate, provided that it does not deadlock or enter an infinite loop. However, due to the absence of blocking constructs in this model, the only source of deadlock is the synchronous method call, and a cyclic chain of such calls is easily detectable at run-time.

The reactive object model has been realized in the language Timber that we will survey in the next section. The model is general enough, though, that we would like to summarize a few informal claims about its properties before we go into language details.

- Reactive objects is a simple and natural model of event-driven systems on various level of detail, from hardware devices to full distributed applications.
- It is a straightforward integration of concurrency and object-oriented programming, with the added bonus of automatic protection of state consistency.
- A single reactive object can easily handle input from multiple asynchronous sources.
- Under assumptions of freedom from non-termination and a very simple form of detectable deadlock, a reactive object is also guaranteed to be responsive in all states.

3 Reactive objects in Timber

We will now give the model of reactive objects a more concrete form, by showing how it is realized in the programming language Timber [4]. Timber is a strongly typed, object-oriented language with constructs specifically aimed at real-time programming; however, its foundation in the reactive object model makes it suitable as a general purpose language as well. We will introduce Timber by a small programming example, after pointing out some distinguishing details.

- In the syntax of Timber, `=` denotes a definition, **let** introduces local definitions, `f x y` is a function `f` applied to two arguments, and `:=` is assignment to a state variable.
- Methods have first-class status: they can be passed as parameters and stored in data structures.
- Commands and declarations are grouped using layout.

Our example is a variant of the program Ping.java from the New I/O API in Java 1.4. This program demonstrates how to concurrently measure the time it takes to connect to a particular TCP port on a number of remote hosts. The Timber source is shown in Figure 1, and the output of a typical run looks as follows:

```
dogbert: 20.018 ms
ratbert: 41.432 ms
ratburg: NetError "lookup failure"
theboss: no response
```

Ping is implemented as an *object template* (i.e., a class). It is parameterized over a list of hosts and a port number, and a record `env` that contains methods for interacting with the environment. Templates define a number of state variables as well as an *interface*, which is typically a record of methods that the environment can invoke. All methods in

```
ping hosts port env =
  template
    outstanding := hosts
  in let
    client host start peer =
      record
        connect      = action
          env.putStrLn(host++": "++show(baseline-start))
          outstanding := remove host outstanding
          peer.close
        neterror err = action
          env.putStrLn(host++": "++show err)
          outstanding := remove host outstanding
        deliver pkt  = action done
        close        = action done
    cleanup = action
      forall h <- outstanding do
        env.putStrLn(h++": no response")
      env.quit
  in record
      main = action
        forall h <- hosts do
          env.inet.tcp.open h port (client h baseline)
        after (2*seconds) cleanup
```

Figure 1. Ping in Timber

this example are asynchronous (as determined by the **action** keyword).

The Ping program is started by creating an instance of the template, and then invoking its `main` method. Internally, Ping objects maintain the state variable `outstanding`, a list of hosts from which a response has not been seen. The main method calls `inet.tcp.open` of the environment in order to initiate a TCP connection to the designated port on every given host (**forall** expresses a loop construct, with `h` as a loop variable). These calls do not wait for the connection to complete, though; instead, the local method `connect` is set up to be invoked when the connection has been established. The record `client` used for this purpose is parameterized over the host we are trying to connect to, the start time of the program, and an environment-provided record containing methods for communicating with the peer host. For timing purposes, actions can refer to the pre-defined variable `baseline`, which is set to the arrival time of the event that triggered the action.

After all connections have been initiated, the asynchronous method `cleanup` is scheduled to be called two seconds after the baseline of `main`. Note that nowhere does the execution of `main` block, it just sets up other actions to react to future events.

All methods of an object can safely manipulate its state variables in a single-threaded fashion. E.g., it is irrelevant here that some methods are defined within the record `client`, and others elsewhere. This flexibility allows us to express a straightforward solution using only one object, with a single state variable. In contrast, the Java program is 287 lines of code, has ten class variables, and needs three threads: one for the demultiplexing of connection events, one for single-threaded printing, and one for timeout.

4 Related work

The reactive object model described in this paper was first developed for the programming language O'Haskell [7]. The current work is, however, our first attempt to distill the programming model as a contribution in its own right. Our language Timber inherits much of its basic design from O'Haskell, but adds several important features, of which the notion of a time-constrained reaction is the most relevant to this paper.

Current work on reactive languages is mostly concerned with the *synchronous* approach to reactivity [3, 5]. The main hypotheses made in this model are that computations take zero time and event delivery is instantaneous. From these assumptions it follows that events received or generated somewhere between two clock ticks are actually occurring *simultaneously*, and hence, for example, multiple invocations of a particular method during an instant must be indistinguishable from just a single invocation.

The unification of the object and process concepts is an idea that stems from the *Actor* model [1, 2]. However, the state of an actor is identified with its current mapping from names to method bodies, and messages to undefined methods are simply queued. Hence actors do not possess the responsiveness property we are emphasizing with our model. Moreover, the Actor model lacks anything similar to our synchronous methods, and asynchronous message delivery is not order-preserving.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[3] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-time Systems. Technical Report 1445, INRIA-Rennes, 1991.

[4] A. P. Black, M. Carlsson, M. P. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. http://www.cse.ogi.edu/PacSoft/projects/Timber/, April 2002.

[5] F. Boussinot, G. Doumenc, and J. Stefani. Reactive Objects. *Annals of Telecommunications*, 51(9-10):459–473, 1996.

[6] P. Chan and R. Lee. *The Java Class Libraries, Second Edition, Volume 2*. Addison-Wesley, 1997.

[7] J. Nordlander. *Reactive Objects and Functional Programming*. PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, May 1999.

[8] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A Note on Distributed Computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Nov. 1994.

[9] J. Zukowski. New I/O Functionality for Java 2 Standard Edition 1.4. http://developer.java.sun.com/developer/technicalArticles/releases/nio/, October 2001.

APPENDIX D

The Semantic Layers of Timber. Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The First Asian Symposium on Programming Languages and Systems (APLAS), Beijing, 2003. Springer-Verlag.

The semantic layers of Timber

Magnus Carlsson[1], Johan Nordlander[2], and Dick Kieburtz[1]

[1] Oregon Health & Science University, {magnus,dick}@cse.ogi.edu
[2] Luleå University of Technology, nordland@sm.luth.se

Abstract. We present a three-layered semantics of *Timber*, a language designed for programming real-time systems in a reactive, object-oriented style. The innermost layer amounts to a traditional deterministic, pure, functional language, around which we formulate a middle layer of concurrent objects, in terms of a monadic transition semantics. The outermost layer, where the language is married to deadline-driven scheduling theory, is where we define message ordering and CPU allocation to actions. Our main contributions are a formalized notion of a *time-constrained reaction*, and a demonstration of how scheduling theory, process calculii, and the lambda calculus can be jointly applied to obtain a direct and succinct semantics of a complex, real-world programming language with well-defined real-time behavior.

1 Introduction

Timber is a new programming language being developed jointly at the Oregon Health & Science University, Chalmers University of Technology, and Luleå University of Technology. The scope of Timber is wide, ranging from low-level device interfaces, over time-constrained embedded systems or interactive applications in general, to very high-level symbolic manipulation and modeling applications. In the context of this text, the distinguishing attributes of the Timber language can be summarized as follows:

- It is based on a programming model of *reactive objects*, that abandons the traditional active request for input in favor of a purely event-driven program structure [15].
- It fully integrates *concurrency* into its semantic foundation, making each object an encapsulated process whose state integrity is automatically preserved.
- It enables *real-time constraints* to be directly expressed in the source code, and opens up the possibility of doing off-line deadline-based schedulability analysis on real code.
- It is a full-fledged pure functional language that achieves referential transparency in the presence of mutable data and non-determinism by means of a monad.
- It upholds *strong static type safety* throughout, even for its message-based, tag-free interprocess communication mechanism.

In this paper we provide a formal semantic definition of Timber. The combination of object-based concurrency, asynchronous reactivity, and purely functional declarativeness is challenging in itself, but we believe it is the existence of a real-time dimension that makes the question of a formal semantics for Timber particularly interesting. The core of our approach is the definition of a *layered* semantics, that separates the semantic concerns in such a way that each layer is meaningful and can be fully understood by referring to just the layer before it. The semantic layers of Timber can be summarized as follows:

- *The functional layer*. This layer amounts to a traditional pure functional language, similar to Haskell [17] or the pure subset of ML [10]. By pure, we mean that computations in this layer do not cause any effects related to state or input/output. We will not discuss this layer very much in this paper, but instead make sure that the other layers do not interfere with it.
- *The reactive layer* is a system of concurrent *objects* with internal state that *react* to messages passed from the environment (input), and in turn send synchronous or asynchronous *messages* to each other, and back to the environment (output). This layer constitutes a form of process calculus that embeds a purely functional sub-language. Although it abstracts away from the flow of real time, the calculus does associate with each message precise, but uninterpreted time constraints.
- *The scheduling layer*. The reactive layer leaves us with a set of messages and objects with time constraints. Some of the messages may compete for the same objects, which in turn are competing for computing resources. The scheduling layer makes precise what the constraints are for the resulting scheduling problem.

The semantic definition has been used to implement an interpreter for Timber, by which we have been able to conduct serious practical evaluations of the language (spanning such diverse areas as GUI toolkit implementation [13] and embedded robot control [7]). A compiler aimed at stand-alone embedded systems is also being developed in tandem with the language specification. We consider the main contributions of this paper to be: (1) a formal definition of the *time-constrained reactions* that give Timber its distinct flavor, and (2) a demonstration of how scheduling theory, process calculii, and the lambda calculus can be jointly applied to obtain a direct and succinct semantics of a complex, real-world programming language with well-defined real-time behavior.

The rest of the paper is organized as follows: Section 2 introduces Timber with an example of an embedded controller. Section 3 constitutes the core of the paper, as it defines the semantics of Timber in three separate subsections corresponding to each semantic layer. Related work is then discussed in Section 4, before we conclude in Section 5.

2 An embedded Timber system

Although Timber is a general-purpose programming language, it has been designed to target *embedded systems* in particular. The software of such a system,

when written in Timber, consists of a number of *objects* which mostly sit and wait for incoming *events* from the physical environment of the embedded system. On the lowest level, events are picked up by peripheral devices, and are transformed into interrupts injected into the CPU. Each interrupt is converted into an *asynchronous message* that has one particular object in the Timber program as its destination. Each message is associated with a *time constraint* in the form of a *baseline* and a *deadline*, both of which are absolute times. The baseline constrains the starting point of a reaction and also functions as a reference point to which time expressions might subsequently relate; it is normally set to the absolute time of the interrupt. The deadline is the time by which the system must have *reacted* to the event for the system to behave correctly. In the tradition of declarative programming , this parameter thus acts as a specification for a correct Timber implementation, ultimately derived from the time constants of the physical environment with which the system interacts.

Once an interrupt has injected a message into a Timber program, the system is no longer in an idle state. The message has *excited* the system, and the destination object starts reacting to the message so that a response can be delivered. As a result, the object may alter its state, and send secondary messages to other objects in the system, some of which may be *synchronous*. Synchronous communication means that the object will rendezvous with the destination object, facilitating two-way communication. Some of the secondary messages generated during the excited state can have baselines and deadlines that are larger than the original, interrupt-triggered message. For example, a car alarm system may react immediately to an event from a motion sensor by turning on the alarm, but also schedule a secondary, delayed reaction that turns off the alarm after a minute.[3] However, eventually the chain reaction caused by an external event will cling out, and the system goes back to an idle state.

Because a Timber system is concurrent, multiple reactions caused by independent external events may be in effect at the same time. It is the job of the presented semantics to specify how the interactions within such a system are supposed to behave.

2.1 Templates, actions and requests

The syntax of Timber is strongly influenced by Haskell [17]. In fact, it stems directly from the object-oriented Haskell extension *O'Haskell*, and can be seen as a successor of that language [12].

To describe Timber more concretely, we present a complete Timber program that implements the car alarm refereed to above in Figure 1. This program is supposed to run on a naked machine, with no other software than the Timber program itself. On such a machine, the environment provides merely two operations, as the definition of the record type `Environment` indicates: one for writing a byte to an I/O port, and one for reading. These methods are synchronous, as indicated by the monadic type constructor `Request`. On the other

[3] A delay perceived as much longer than a minute, though!

hand, the interface a naked program presents to its environment will merely consist of handlers for (a subset of) the interrupts supported by the hardware. This is expressed as a list of asynchronous methods (actions) paired with interrupt numbers, as the type `Program` prescribes.

```
record Environment where
   write      :: Port -> Byte -> Request ()
   read       :: Port -> Request Byte

record Program where
   irqvector  :: [(Irq,Action)]

alarm :: Environment -> Template Program
alarm env =
  template
     trigged := True
  in let
     moved = before (100*milliseconds) action
        if trigged then
           env.write siren 1
           trigged := False
           after (1*minutes) turnoff
           after (10*minutes) enable
     turnoff = action
        env.write siren 0
     enable = action
        trigged := True
  in record
     irqvector = [(motionsensor,moved)]
```

Fig. 1. The car alarm program.

A Timber program is a function from the program environment to a *template* that returns an interface to an object. At boot time, the template `alarm` will be executed, thereby creating one instance of an alarm handling object, whose interface is the desired interrupt vector.

We assume that there is a particular I/O port `siren`, whose least significant bit controls the car siren, and that the interrupt number `motionsensor` is associated with the event generated by a motion sensor on the car.

The definition of `alarm` reveals a template for an object with one internal state variable `trigged`, initialized to `True`. The purpose of `trigged`, as we will see, is to ensure that once the alarm goes off, there will be a grace period until it is possible to trigger the alarm again.

The returned interrupt vector associates the action `moved` with the motion sensor interrupt. When `moved` is invoked, and if the alarm is trigged, it will turn on the siren, and set `trigger` to false. It will also invoke two local asynchronous methods, each one with a lower time bound on the actual message reception. The first message, which is scheduled to be handled one minute after the motion sensor event, simply turns off the siren. The second message, which will arrive after ten minutes, re-enables the alarm so that it may go off again.

By means of the keyword **before**, action `moved` is declared to have a deadline of one tenth of a second. This means that a correct implementation of the program is required to turn on the alarm within 100 milliseconds after the motion sensor event. This deadline also carries over to the secondary actions, which for example means that the alarm will shut off no later than one minute and 0.1 seconds after the motion sensor event (and no earlier than one minute after the event, by the lower time bound).

The handling of delays in our example captures a characterizing aspect of the reactive semantics of Timber. Instead of temporarily halting execution at some traditional "sleep" statement, our method `moved` will terminate as quickly as the CPU allows, leaving the alarm object free to react to any pending or subsequent calls to `moved`, even while the 1 and 10 minute delay phases are active. A degraded motion sensor generating bursts of interrupts will thus be handled gracefully by our program; something that would require considerably more work in a language where events are queued and event-handling can be arbitrarily delayed.

```
alarm env = do
  self <- new True
  let moved = act self ⟨0,100*milliseconds⟩
                (do trigged <- get
                    if trigged then do
                       env.write sirenport 1
                       set False
                       aft (1*minutes) turnoff
                       aft (10*minutes) enable
                    else return () )
     turnoff = act self ⟨0,0⟩
                (env.write sirenport 0)
     enable = act self ⟨0,0⟩
                (set True)
  return ( record irqs = [(motionsensor,moved)] )
```

Fig. 2. The car alarm program, desugared.

2.2 The O monad

```
return:: a -> O s a
(>>=) :: O s a -> (a -> O s b) -> O s b
handle:: O s a -> (Error -> O s a) -> O s a
raise :: Error -> O s a
bef   :: Time -> O s a -> O s a
aft   :: Time -> O s a -> O s a
set   :: s -> O s ()
get   :: O s s
new   :: s -> O s' (Ref s)
act   :: Ref s -> (Time,Time) -> O s a -> O s' Msg
abort :: Msg -> O s ()
req   :: Ref s -> O s a -> O s' a
```

Fig. 3. The constants in the O monad.

The Timber constructs **template**, **action**, and **request** are nothing but syntactic sugar for monadic computations that take place in the `O` monad, as described in [14, 12]. The type `O` s a stands for a computation that can be executed inside an object whose local state has type s, resulting in a value of type a. In the full Timber language, *polymorphic subtyping* is applied to give constants that are independent of the local state a more flexible type (see for example the types `Action` and `Request` mentioned above) [13]. However, we will ignore this issue in the following presentation, as it has no operational implications.

A desugared version of the alarm program is given in Figure 2. Strictly speaking, the program is not a literate result of the desugaring rules—for readability, we introduced the do-notation and performed some cosmetic simplifications.

The desugared program refers to a number of primitive operations in the O monad, whose type signature are given in Figure 3. An informal meaning of these constants can be obtained by comparing the desugared program in Figure 2 with the original. Their precise formal meaning will be the subject of the next section. The actual desugaring rules are given in Appendix A.

3 The semantic layers

3.1 The functional layer

We will not specify very much of the functional layer in this presentation; instead we will view it as a strength of our layered approach that so much of the first layer can be left unspecified. One of the benefits with a monadic marriage of effects and evaluation is that it is independent of the evaluation semantics—this property has even been proposed as the definition of what *purely functional* means in the presence of effects [19].

$$
\begin{aligned}
\texttt{return}\ e_1\ \texttt{>>=}\ e_2 &\mapsto e_2\ e_1\\
\texttt{raise}\ e_1\ \texttt{>>=}\ e_2 &\mapsto \texttt{raise}\ e_1\\
\texttt{return}\ e_1\ \texttt{`handle`}\ e_2 &\mapsto \texttt{return}\ e_1\\
\texttt{raise}\ e_1\ \texttt{`handle`}\ e_2 &\mapsto e_2\ e_1\\
\texttt{bef}\ d'\ (\texttt{act}\ n\ \langle b, d\rangle\ e) &\mapsto \texttt{act}\ n\ \langle b, d'\rangle\ e\\
\texttt{aft}\ b'\ (\texttt{act}\ n\ \langle b, d\rangle\ e) &\mapsto \texttt{act}\ n\ \langle b', d\rangle\ e
\end{aligned}
$$

Fig. 4. Functional layer: reduction rules.

That said, it is also the case that a lazy semantics in the style of Haskell adds extra difficulties to the time and space analysis of programs, something which is of increased importance in the construction of real-time and embedded software. For this reason, we are actually shifting to a strict semantics for Timber, which is a clear breach with the tradition of its predecessors. Still, strictness should not be confused with impurity, and to underline this distinction, we will assume a very general semantics that is open to both both lazy and strict interpretation in this paper. We hope to be able to report on the specifics of strictness in Timber in later work, especially concerning the treatment of recursive values and imprecise exceptions [4, 11, 8].

We assume that there is a language $\mathcal{E}$ of expressions that includes the pure lambda calculus, and whose semantics is given in terms of a small-step evaluation relation $\mapsto$. Let e range over such expressions. Moreover, we assume that the expressions of $\mathcal{E}$ can be assigned types by means of judgments of the form $\Gamma \vdash e : \tau$, where τ ranges over the types of $\mathcal{E}$, and Γ stands for typing assumptions.

A concrete example of what $\mathcal{E}$ and $\mapsto$ might look like can be found in [12]. There a semantics is described that allows concurrent reduction of all redeces in an expression, even under a lambda. We will neither assume nor preclude such a general semantics here, but for the purpose of proving type soundness of the reaction layer, we require at least the following property:

Theorem 1 (Subject reduction). *If $\Gamma \vdash e : \tau$ and $e \mapsto e'$, then $\Gamma \vdash e' : \tau$.*

In order to connect this language with the reactive semantics of the next section, we extend $\mathcal{E}$ with the constants of Figure 3, and the extra evaluation rules of Figure 4.

The first four constants listed in Figure 3 constitute a standard exception-handling monad, for which meaning is given by the first four rules of Figure 4. Likewise, the constants `bef` and `aft` merely serve to modify the time constraint argument of the `act` constant, so their full meaning is also defined in the functional layer.

The remaining constants, however, are treated as *uninterpreted constructors* by the functional semantics—it is giving meaning to those constants that is the purpose of the reactive semantic layer. Similarly, concrete realization of the type constructors `O`, `Ref` and `Msg` is postponed until the reactive layer is defined; as far as the functional layer is concerned, `O`, `Ref` and `Msg` are just opaque, abstract types.

3.2 The reactive layer

To give semantics to the reactive layer, we regard a running Timber program as a system of concurrent processes, defined by the grammar in Figure 5. A primitive process is either

- an *empty message*, tagged with the name m. This is the remnant of a message that is either delivered or aborted;
- a *message* tagged with m, carrying the code e, to be executed by an object with name n, obeying the time constraint c. In the case of a synchronous message, it also carries a *continuation* K, which amounts to a suspended requesting object (see below);
- an *idle object* with identity n, which maintains its state s awaiting activation by messages;
- an *active object* with identity n, state s, executing the code e with the time constraint c. In case the object is servicing a synchronous request, the continuation K amounts to the suspended requesting object.

$P ::= \langle\rangle_m$		Empty message
	$\langle n, e, K\rangle^c_m$	Pending message
	$(\!\lvert s \rvert\!)_n$	Idle object
	$(\!\lvert s, e, K \rvert\!)^c_n$	Active object
	$P \parallel P$	Parallel composition
	$\nu n.\, P$	Restriction

Fig. 5. The process grammar.

Finally, processes can be composed in parallel, and the scope of names used for object identities and message tags can be restricted. We will assume that parallel composition has precedence over the restriction operator, which therefore always extend as far to the right as possible.

Continuations are requesting objects that are waiting for a request (synchronous message) to finish. Since a requesting object can in turn be servicing requests from other objects, continuations are defined recursively, according to the following grammar that denotes a *requesting object* or an *empty continuation*:

$$K ::= (\!\lvert s, \mathcal{M}, K \rvert\!)_n \mid 0$$

A requesting object with identity n contains the state s, and a *reaction context* $\mathcal{M}$, which is an expression with a hole, waiting to be filled by the result of the

request. Just as for normal objects, there is a continuation K as well, in case the requesting object is servicing a request from another object. For asynchronous messages, and objects that are not servicing requests, the continuation is empty.

Reaction contexts are used in two ways in the reactive layer. As we have already seen, they are used in an unsaturated way to denote expressions with holes in requesting objects. They are also used in the following section to pinpoint where in an expression the reaction rules operate. Reaction contexts are given by the grammar

$$\mathcal{M} ::= \mathcal{M} \texttt{ >>= } e \mid \mathcal{M} \texttt{ `handle` } e \mid [\,] \ .$$

The structural congruence relation In order for the relation to bring relevant process terms together for the reaction rules given in the next section, we assume that processes can be rearranged by the *structural congruence* $\equiv$,

Associativity	$P_1 \parallel (P_2 \parallel P_3) \equiv (P_1 \parallel P_2) \parallel P_3$
Symmetry	$P_1 \parallel P_2 \equiv P_2 \parallel P_1$
Scope extension	$P_1 \parallel \nu n.P_2 \equiv \nu n.P_1 \parallel P_2$ if $n \notin fv(P_1)$
Scope commutativity	$\nu n.\nu m.P \equiv \nu m.\nu n.P$
Renaming	$\nu n.P \equiv \nu m.[{}^{m}/_{n}]P$ if $m \notin fv(P_1)$
Inert message	$P \parallel \nu m.\langle\rangle_m \equiv P$
Inert object	$P \parallel \nu n.(\!|s|\!)_n \equiv P$

Fig. 6. The structural congruence.

induced by the elements in Figure 6. These allow for the process terms to be rearranged and renamed, so that *e.g.* a message and its destination object can be juxtaposed for the relevant rule to apply. The last two elements of the equivalence allow us to garbage collect inert objects or messages that cannot possibly interact with any other process in the system. More specifically, an idle object whose identity is unknown to the rest of the system cannot possibly receive any message. Similarly, an empty message whose tag is "forgotten" can be eliminated.

The reactive relation The reactive layer of our semantics consists of a *reaction relation* $\longrightarrow$, that defines how objects interact with messages and alter internal state. The reaction relation is characterized by the axioms shown in Figure 7, which together with the structural rules in Figure 8 define $\longrightarrow$ for general process terms. Most of the axioms use a reduction context to pinpoint the next relevant O monad constant inside an active object.

Rules Set and Get allows for the local state of an object to be written or read. The next three rules introduce a fresh name on the right-hand side by using restriction, and we tacitly assume here that the names m, n' do not occur free in the process term on the left-hand side.

Rule New defines creation a new, idle object whose identity is returned to the creator, and with an initial state as specified by the argument to the constant `new`.

$$
\begin{array}{lrl}
\textsc{Set} & (\!|s, \mathcal{M}[\texttt{set}\ e], K|\!)^c_n & \longrightarrow (\!|e, \mathcal{M}[\texttt{return}\ ()], K|\!)^c_n \\
\textsc{Get} & (\!|s, \mathcal{M}[\texttt{get}], K|\!)^c_n & \longrightarrow (\!|s, \mathcal{M}[\texttt{return}\ s], K|\!)^c_n \\
\textsc{New} & (\!|s, \mathcal{M}[\texttt{new}\ e], K|\!)^c_n & \longrightarrow \nu n'.\, (\!|s, \mathcal{M}[\texttt{return}\ n'], K|\!)^c_n \parallel (\!|e|\!)_{n'} \\
\textsc{Act} & (\!|s, \mathcal{M}[\texttt{act}\ n'\ d\ e], K|\!)^c_n & \longrightarrow \nu m.\, (\!|s, \mathcal{M}[\texttt{return}\ m], K|\!)^c_n \parallel \langle n', e, 0\rangle^{c+d}_m \\
\textsc{Req} & (\!|s, \mathcal{M}[\texttt{req}\ n'\ e], K|\!)^c_n & \longrightarrow \nu m.\, \langle n', e, (\!|s, \mathcal{M}, K|\!)_n\rangle^c_m \\
\textsc{Run} & (\!|s|\!)_n \parallel \langle n, e, K\rangle^c_m & \longrightarrow (\!|s, e, K|\!)^c_n \parallel \langle\rangle_m \\
\textsc{Done} & (\!|s, r\ e, 0|\!)^c_n & \longrightarrow (\!|s|\!)_n \quad \textit{where } r \in \{\texttt{raise}, \texttt{return}\} \\
\textsc{Rep} & (\!|s, r\ e, (\!|s', \mathcal{M}, K|\!)_m|\!)^c_n & \longrightarrow (\!|s|\!)_n \parallel (\!|s', \mathcal{M}[r\ e], K|\!)^c_m \\
 & & \quad \textit{where } r \in \{\texttt{raise}, \texttt{return}\} \\
\textsc{Abort} & (\!|s, \mathcal{M}[\texttt{abort}\ m], K|\!)^c_n \parallel P & \longrightarrow (\!|s, \mathcal{M}[\texttt{return}\ ()], K|\!)^c_n \parallel \langle\rangle_m \\
 & & \quad \textit{where } P \in \{\langle n', e, 0\rangle^{c'}_m, \langle\rangle_m\}
\end{array}
$$

Fig. 7. Reactive layer: axioms.

In rule Act, an object is sending an asynchronous message with code e to a destination object n'. The time constraint, or *timeline*, of a message is computed by adding the *relative* time constraint d to the constraint of the sending object. The addition of time constraints is described in more detail in the section 3.2. Note that, as a consequence of the substitution-based evaluation semantics we have assumed, messages contain the actual code to be run by destination objects, instead of just method names. This should not be confused with breaking the abstraction barrier of objects, since object interfaces normally expose only action and request values, not the object identifiers themselves. Without knowledge of an object's identity, it is impossible to send arbitrary code to it.

$$
\frac{P \equiv P' \qquad P' \longrightarrow Q' \qquad Q' \equiv Q}{P \longrightarrow Q}\ \textsc{Equiv}
$$

$$
\frac{P \longrightarrow P'}{P \parallel Q \longrightarrow P' \parallel Q}\ \textsc{Par}
$$

$$
\frac{P \longrightarrow P'}{\nu n.\, P \longrightarrow \nu n.\, P'}\ \textsc{Res}
$$

Fig. 8. Reactive layer: structural rules.

Rule Req forms a synchronous message, by suspending the requesting object and embedding it as a continuation within the message. Here the unsaturated context $\mathcal{M}$ of the caller is saved, so that it can eventually be filled with the result of the request. The time constraint of a synchronous message is the same as that of the requesting object—the intuition here is that a request is analogous to a function call, and that servicing such a call can be neither more nor less urgent than the computation of the caller.

In rule RUN, an idle object is juxtaposed to a message with matching destination. The "payload" of the message (the code, continuation and time constraint) is extracted from the message, which is left empty. This rule forms the essence of what constitutes a reaction in Timber.

When an active object eventually reaches a terminating state (as represented by a code expression of the form `return` e or `raise` e), the action taken depends on its continuation. Rule DONE specifies that an object executing an asynchronous message just silently enters the idle state, where it will be ready to accept new messages. An object occupied by a request, on the other hand, also needs to reply to the requesting object. This is handled in the rule REP, in which the waiting caller context $\mathcal{M}$ is applied to the reply (which may be an exception), and the continuation is released as an active object again. At the same time, the servicing object turns idle, ready for new messages.

The ABORT rule shows how a pending message can be turned into an empty one before delivery, thus effectively removing it from the system. If the destination object has already started executing the message, or if the message was previously aborted, the rule will match against an empty message instead, leaving it unaffected.

Finally, there is a rule EVAL that connects the functional and reactive layers, by promoting evaluation to reaction:

$$\frac{s \mapsto s' \quad e \mapsto e'}{(\!|s, \mathcal{M}[e], K|\!)^c_n \longrightarrow (\!|s', \mathcal{M}[e'], K|\!)^c_n} \text{ EVAL}$$

Note that we allow for the concurrent evaluation both the state and code components of an object here, although with a strict functional layer, the state component will of course already be a value.

Timeline arithmetic The time constraint, or *timeline*, of an asynchronous message is obtained by adding the time constraint $\langle b, d\rangle$ of the sending object to the *relative* timeline $\langle \beta, \delta\rangle$ supplied to the constructor `act`. This operation is defined as follows:

$$\begin{aligned}\langle b, d\rangle + \langle \beta, \delta\rangle &= \langle max(b, b+\beta), max(d, b+\beta+\delta)\rangle \quad \text{if } \delta > 0\\ &= \langle max(b, b+\beta), max(d, d+\beta)\rangle \qquad \text{if } \delta \leq 0\end{aligned}$$

The maximum functions used here serve to ensure that the timeline of a message cannot be tighter than the timeline of the sending object; *i.e.*, both the baseline and the deadline of a message must be at least as large as those of the sender. This prevents the introduction of paradoxical situations where servicing a secondary reaction would be more urgent than executing the code that caused it.

As can be seen, a special case occurs when the relative deadline of a message is zero or below; then the deadline of the sender is kept but interpreted relative to the new baseline. This is how the two delayed messages sent in the `moved` method in Figure 2 are assigned their deadlines of 1 minute + 100 milliseconds, and 10 minutes + 100 milliseconds, respectively.

Note that this exception is added purely for the convenience of the programmer, who would otherwise always have to specify an explicit deadline whenever a new baseline is given. Note also that a relative deadline of zero amounts to a timeline that cannot be satisfied by any kind of finite computing resource, so an exception for this value does not really limit the expressiveness for the programmer.

Deadlock A Timber program that only uses asynchronous communication is guaranteed to be free from deadlock; however, since the sender of a synchronous message is unresponsive while it waits for a reply, the potential of deadlock arises. On the other hand, unlike many other languages and interprocess communication mechanisms, Timber allows for really cheap detection of deadlock. What is required is that each object keeps a pointer to the servicing object as long as it is blocked in a request, and that that the `req` operation starts by checking that setting up such a pointer will not result in cycle. If this is indeed the case, `req` results in an exception.

Preferably, our reaction semantics should formalize this behavior, as it is of utmost importance for the correctness of systems in which deadlock can occur. Unfortunately, our recursive definition of continuations actually denotes a linked structure that points in the other direction; from a server to its current client, if any. Duplicating this structure with links going in the forward direction makes the reaction axioms look extremely clumsy, and we have not found the formal definition of deadlock detection valuable enough to outweigh its cost. Instead we supplement the reaction axioms of Figure 7 with an informal rule that reads

$$(\!|s, \mathcal{M}[\texttt{req}\ n'\ e], K|\!)^c_n \longrightarrow (\!|s, \mathcal{M}[\texttt{raise Deadlock}], K|\!)^c_n$$

if the sending object n is found by following the forward pointers starting at n'. It is our hope that a neater way of specifying this behavior can eventually be developed.

Properties of the reactive layer Analogous to the subject reduction property that we assume for the functional layer, we establish what we may call a *subject reaction* property for our process calculus; *i.e.*, the property that all reaction rules preserve well typedness of processes. Well-typedness is defined by the straightforward typing judgments of Appendix B, and the subject reaction theorem looks as follows:

Theorem 2 (Subject Reaction). *If* $\Gamma \vdash P$ well-typed *and* $P \longrightarrow Q$*, then* $\Gamma \vdash Q$ well-typed

An attractive property of our layered semantics is that reduction and reaction live in independent worlds. There are no side effects in the functional layer, its only role is to enable rules in the deterministic layer, and further reduction cannot retract any choices already enabled. This can be captured in a diamond property that we call *functional soundness*, which says that the $\mapsto$ and $\longrightarrow$ relations commute.

Let $\underset{\textsc{Eval}}{\longrightarrow}$ be the relation formed by the structural rules in Figure 8 and rule Eval, but none of the axioms in Figure 7. Let $\underset{\neg\textsc{Eval}}{\longrightarrow}$ be the reaction relation formed by including the structural rules and the axioms, but not rule Eval.

Theorem 3 (Functional soundness). *If* $P \underset{\neg\textsc{Eval}}{\longrightarrow} Q$ *and* $P \underset{\textsc{Eval}}{\longrightarrow} P'$*, then there is a* Q' *such that* $P' \underset{\neg\textsc{Eval}}{\longrightarrow} Q'$ *and* $Q \underset{\textsc{Eval}}{\longrightarrow} Q'$*.*

The reader is referred to [12] for more elaboration and proofs of the above properties.

3.3 The scheduling layer

The reactive layer leaves us with a system that is highly non-deterministic—it says nothing about in which order objects should run messages, or in which order concurrently active objects may progress. The scheduling layer puts some extra constraints on the system, by consulting the hitherto uninterpreted time constraints attached to messages and objects. The requirements on the scheduler are formulated in terms of a *real-time trace*.

Definition 1 (Real-time trace). *A* real-time transition *is a transition* $P \longrightarrow Q$ *associated with a value* t *of some totally ordered time domain, written* $P \xrightarrow{t} Q$*. A* real-time trace $\mathcal{T}$ *is a possibly infinite sequence of real-time transitions* $P_i \xrightarrow{t_i} P_{i+1}$ *such that* $t_i < t_{i+1}$*. We call each* P_i *in a real-time trace a* trace state.

A real-time trace thus represents the execution of a Timber program on some specific machine, with machine-specific real-time behavior embedded in the t_i. Our first goal is to define which real-time behaviors are acceptable with respect to the time-constraints of a program.

Definition 2 (Transition timeline). *Let the timeline of a reaction axiom be the pair* c *as it occurs in the reaction rules of Figure 7, and let the timeline of a structural reaction rule be the timeline of its single reaction premise. We now define the timeline of a transition* $P \xrightarrow{t} Q$ *as the timeline of the rule used to derive* $P \longrightarrow Q$*.*

This definition leads to the notion of *timeliness*:

Definition 3 (Timeliness). *A real-time transition* $P \xrightarrow{t} Q$ *with timeline* $\langle b, d\rangle$ *is* timely *iff* $b \leq t \leq d$*. A real-time trace* $\mathcal{T}$ *is timely iff every transition in* $\mathcal{T}$ *is timely.*

Our second goal is to constrain the dispatching of messages to occur in priority order. First we need to formalize the notion of a dispatch:

Definition 4 (Dispatch). *A* dispatch *from* $\mathcal{T}$ *is a real-time transition* $P \xrightarrow{t} P'$ *derived from rule* Run*, such that* P *is the final trace state of* $\mathcal{T}$*. An* (n, Q)-dispatch *from* $\mathcal{T}$ *is a dispatch from* $\mathcal{T}$ *where* n *is the name of the activated object, and* Q *is the dispatched message.*

The intention is that whenever there are several possible dispatches from some $\mathcal{T}$, the semantics should prescribe which one is chosen. To this end we will need to define an ordering relation between messages.

First, we let timelines be sorted primarily according to their deadlines.

Definition 5 (Timeline ordering). $\langle b, d\rangle < \langle b, d\rangle$ *iff* $d < d'$, *or* $d = d'$ *and* $b < b'$.

Second, we want to induce an ordering on messages whose timelines are identical. We will assume, without loss of generality, that ν-bound names are chosen to be unique in a trace state, and that the renaming congruence rule is never applied in an EQUIV transition. This makes it possible to uniquely determine the time when a bound name appears in a trace.

Definition 6 (Binding times). *The* binding time $bt(n, \mathcal{T})$ *of a name* n *in a real-time trace* $\mathcal{T}$ *is the largest time* t_i *associated with a real-time transition* $P_i \xrightarrow{t_i} P_{i+1}$ *in* $\mathcal{T}$, *such that* P_{i+1} *contains a* ν*-binder for* n *but* P_i *does not.*

From the previous two definitions we can construct a partial order on pending messages.

Definition 7 (Message ordering). $\langle n, e, K\rangle^c_m <_{\mathcal{T}} \langle n, e', K'\rangle^{c'}_{m'}$ *iff* $c < c'$, *or* $c = c'$ *and* $bt(m, \mathcal{T}) < bt(m', \mathcal{T})$

Note that this definition only relates messages aimed for some particular object, and that these messages are actually totally ordered.

From the ordering of messages follows the notion of a *minimal* dispatch.

Definition 8 (Minimal dispatch). *An* (n, Q)*-dispatch from* $\mathcal{T}$ *is* minimal *iff, for all possible* (n, Q')*-dispatches from* $\mathcal{T}$, $Q <_{\mathcal{T}} Q'$. *A dispatch from* $\mathcal{T}$ *is minimal if it is a minimal* (n, Q)*-dispatch from* $\mathcal{T}$ *for some* n *and* Q.

Finally, the timeliness and minimality constraints can be applied to real-time traces in order to identify the *valid* ones.

Definition 9 (Valid trace). *A real-time trace* $\mathcal{T}$ *is* valid *iff* $\mathcal{T}$ *is timely and every dispatch from a prefix* $\mathcal{T}'$ *of* $\mathcal{T}$ *is minimal.*

The real-time semantics of a Timber program is thus the set of valid traces it generates.

Some notes regarding these definitions:

1. The notion of minimality makes message dispatching fully deterministic. What the semantics prescribes is actually a *priority queue* of messages for each object, that resorts to FIFO order when priorities (timelines) are identical. This is also how our Timber compiler implements the semantics. Our reasons for defining message queuing here and not in the reactive layer are twofold: First, the complexity that arises from maintaining explicit queues in the process calculus is daunting. Second, leaving out ordering concerns is more in line with the process calculus tradition, and allows for easier comparison between Timber and other languages based on similar formalisms, like Concurrent Haskell.

2. All scheduling flexibility is captured in the selection of which object to run—because message order is fixed, the reaction axioms of Figure 7 do not offer any flexibility in choosing the next transition for a particular object. On the other hand, whenever there are several objects capable of making a timely transition, the semantics allows any one of them to be chosen. This opens up for pre-emptive scheduling, and coincides with our intuition that objects execute in parallel, but are internally sequential.
3. It follows from the monotonicity of real time that if a dispatch meets the baseline constraint of its timeliness requirement, all transitions involving the same object up to its next idle state will also meet the baseline constraint. Likewise, if an object becomes idle by means of a DONE or REP transition that meets its deadline, all transitions involving this object since it was last idle must also have met this deadline.
4. Meeting the baseline constraint of a dispatch is always feasible; it just amounts to refraining from making a certain transition. This can easily be implemented by putting messages on hold in a system-wide timer queue until their baselines have passed. On the other hand, meeting a deadline constraint is always going to be a fight against time and finite resources. Statically determining whether the execution of a Timber program will give rise to a valid trace in this respect is in general infeasible; however, we note that scheduling theory and feasibility analysis is available for attacking this problem, at least for a restricted set of Timber programs.
5. It can be argued that a Timber implementation should be able to continue execution, even if the deadline constraint of the timeliness requirement cannot be met for some part of its trace. Indeed, this is also what our Timber compiler currently does. However, it is not clear what the best way of achieving deadline fault tolerance would be, so we take the conservative route and let the Timber semantics specify only the desired program behavior at this stage.

On a uni-processor system, the scheduling problems generated by our semantics bear an attractively close resemblance to the problems addressed by deadline-based scheduling theory [21]. In fact, the well-known optimality result for fully preemptive Earliest-Deadline-First scheduling [3] can be directly recast to the Timber setting as follows:

Theorem 4 (Optimality of EDF). *For a given real-time trace, let the execution time attributed to a transition only depend on the reaction axiom from which the transition is derived. Moreover, let a re-ordering of the trace be the result of repeatedly applying the equivalence* $P \parallel Q \longrightarrow P' \parallel Q \longrightarrow P' \parallel Q' \equiv P \parallel Q \longrightarrow P \parallel Q' \longrightarrow P' \parallel Q'$ *(structurally lifted to transitions).*

Then, if there exists a re-ordering of transitions that results in a timely trace, re-ordering the transitions according to the principle of EDF will also result in a timely trace.

It is our intention to study this correspondence in considerable more detail, especially how the presence of baseline constraints affects existing feasibility

theory. However, it should also be noted that the scheduling layer semantics does not *prescribe* EDF scheduling. In particular, static schedules produced by off-line simulations of a program is an interesting alternative we are also looking into [9].

4 Related work

The actions in Timber resemble the *tasks* in the E machine [5], where the programmer can specify that a reaction to an event should be delivered precisely at a certain point in time. Consequently, the output of a task will be queued until it is time to react, and the E machine becomes a deterministic system, in sharp contrast to Timber. Similarly, the language H [22] is a Haskell-like language where functions over *timestamped messages* are interconnected through ports. For an input message, a timestamp indicates the time of an event, and for output, it specifies exactly when the message should be output from the system.

In the *synchronous* programming school (Esterel, Signal, Lustre), programs are usually conducted by one or more periodic clocks, and computations are assumed to terminate within a clock period [2]. In contrast, Timber does not make any assumptions about periodic behavior, even though it shares the concept of reactivity with the synchronous languages.

The UDP Calculus [20, 25] provides a formalism for expressing detailed behavior of distributed systems that communicate via the UDP protocol. Part of the structure of our process calculus can be found in the UDP Calculus as well; for example the representation of hosts (objects) as process terms tagged with unique names, and the modelling of messages as free-floating terms in their own right. The UDP Calculus has a basic notion of time constraints in the shape of terms annotated with timers, although at present this facility is only meant to model message propagation delays and timeout options. Moreover, the focus on actual UDP implementations in existing operating systems makes the UDP Calculus more narrow in scope than Timber, but also significantly more complex: the number of transition axioms in the UDP Calculus is 78, for example, where we get away with about 10.

The language Hume [18] has similar design motives as Timber: it has asynchronous, concurrent processes, is based on a pure functional core, and targets embedded, time-critical systems. However, Hume is an example of languages that identify the concept of *real time* with bounds on resources, not with means for specifying time constraints for the delivery of reactions to events. Apart from Hume, this group of languages also include Real-Time FRP and Embedded ML [24, 6].

On a historical note, Embedded Gofer [23] is an extension to a Haskell precursor Haskell aimed at supporting embedded systems. It has an incremental garbage collector, and direct access to interrupts and I/O ports, but lacks any internal notion of time. The same can also be said for Erlang [1]. Although it has been successfully applied in large real-time telecom switching systems, it

only provides best-effort, fixed-priority scheduling, and lacks a static type-safety property.

The technique of separating a semantics into a purely functional layer and an effectful process calculus layer has been used in the definition of Concurrent Haskell [16] and O'Haskell [14]. Although it is well known that a functional language can be encoded in process calculii, such an encoding would obscure the semantic stratification we wish to emphasize.

5 Conclusions and future work

We have given a semantics for Timber, stratified into independent layers for functional evaluation, object and message reactions, and time-sensitive scheduling. The language is implemented in terms of an full-featured interpreter, and we are currently developing a compiler that generates C code for targeting embedded systems. In the near future, we plan to apply and implement deadline-based scheduling analysis techniques for Timber, and nail down the specifics of shifting to a strict semantics in the functional layer.

6 Acknowledgments

The design of the Timber language has been influenced by contributions by many people within the Timber group at OHSU; Iavor Diatchki and Mark Jones in particular have been active in the development of Timber's time-constrained reactions. We would also like to thank members of the PacSoft research group at OHSU for valuable feedback, notably Thomas Hallgren.

References

1. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
2. A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.
3. M. Dertouzos. Control Robotics: the Procedural Control of Physical Processes. *Information Processing*, 74, 1974.
4. L. Erkök and J. Launchbury. Recursive monadic bindings. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP'00*, pages 174–185. ACM Press, September 2000.
5. T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.
6. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

7. M. P. Jones, M. Carlsson, and J. Nordlander. Composed, and in control: Programming the Timber robot. `http://www.cse.ogi.edu/~mpj/timbot/ComposedAndInControl.pdf`, 2002.
8. S. L. P. Jones, A. Reid, F. Henderson, C. A. R. Hoare, and S. Marlow. A semantics for imprecise exceptions. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 25–36, 1999.
9. R. Kieburtz. Real-time reactive programming for embedded controllers. `ftp://cse.ogi.edu/pub/pacsoft/papers/timed.ps`, 2001.
10. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
11. E. Moggi and A. Sabry. An abstract monadic semantics for value recursion. In *Workshop on Fixed Points in Computer Science*, 2003.
12. J. Nordlander. *Reactive Objects and Functional Programming*. Phd thesis, Department of Computer Science, Chalmers University of Technology, Gothenburg, 1999.
13. J. Nordlander. Polymorphic subtyping in O'Haskell. *Science of Computer Programming*, 43(2-3), 2002.
14. J. Nordlander and M. Carlsson. Reactive Objects in a Functional Language – An escape from the evil "I". In *Proceedings of the Haskell Workshop*, Amsterdam, Holland, 1997.
15. J. Nordlander, M. Jones, M. Carlsson, D. Kieburtz, and A. Black. Reactive objects. In *The Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, 2002.
16. S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *ACM Principles of Programming Languages*, pages 295–308, St Petersburg, FL, Jan. 1996. ACM Press.
17. S. Peyton Jones et al. Report on the programming language Haskell 98, a non-strict, purely functional language. `http://haskell.org`, February 1999.
18. A. Rebón Portillo, K. Hammond, H.-W. Loidl, and P. Vasconcelos. Granularity analysis using automatic size and time cost inference. In *Proceedings of IFL '02—Implementation of Functional Languages*. Springer Verlag, September 2002.
19. A. Sabry. What is a Purely Functional Language? *Journal of Functional Programming*, 8(1):1–22, 1998.
20. A. Serjantov, P. Sewell, and K. Wansbrough. The UDP Calculus: Rigorous Semantics for Real Networking. In *Theoretical Aspects of Computer Software, 4th International Symposium, TACS 2001*, Sendai, Japan, Oct 2001.
21. J. Stankovich, editor. *Deadline Scheduling for Real-time Systems, EDF and Related Algorithms*. Kluwer, 1998.
22. S. Truvé. A new H for real-time programming. `http://www.cs.chalmers.se/~truve/NewH.ps`.
23. M. Wallace and C. Runciman. Lambdas in the liftshaft - functional programming and an embedded architecture. In *Functional Programming Languages and Computer Architecture*, pages 249–258, 1995.
24. Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.
25. K. Wansbrough, M. Norrish, P. Sewell, and A. Serjantov. Timing UDP: mechanized semantics for sockets, threads and failures. In *11th European Symposium on Programming, ESOP 2002*, Grenoble, France, April 2002.

A Desugaring

$$
\begin{aligned}
[\![\textbf{do } c]\!]_v &= [\![c]\!]_v \\
[\![\textbf{do } p \texttt{ <- } e\texttt{;}\ cs]\!]_v &= [\![e]\!]_v \texttt{ >>= \ } p \texttt{ -> } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{do } c\texttt{;}\ cs]\!]_v &= [\![c]\!]_v \texttt{ >> } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{do let } ds\texttt{;}\ cs]\!]_v &= [\![\textbf{let } ds \textbf{ in do } cs]\!]_v \\
[\![\textbf{template } as \textbf{ in } e]\!]_v &= \left[\!\!\left[\begin{array}{l} \textbf{do } \texttt{self <- new } es \\ \quad \texttt{return } e \end{array} \right]\!\!\right]_{v'} \\
\text{where } &\begin{cases} v' = [\ x \mid x \texttt{ := } e \leftarrow as\ ] \\ es = [\ e \mid x \texttt{ := } e \leftarrow as\ ] \end{cases} \\
[\![\textbf{action } cs]\!]_v &= \texttt{act self } \langle 0,0 \rangle\ [\![\textbf{do } cs]\!]_v \\
[\![\textbf{request } cs]\!]_v &= \texttt{req self } [\![\textbf{do } cs]\!]_v \\
[\![\textbf{before}]\!]_v &= \texttt{bef} \\
[\![\textbf{after}]\!]_v &= \texttt{aft}
\end{aligned}
\qquad
\begin{aligned}
[\![p \texttt{ := } e]\!]_v &= [\![\texttt{set } ((\texttt{\textbackslash } p \texttt{ -> } v)\ e)]\!]_v \\
&\quad \text{if } fv(p) \subseteq v \\
[\![e]\!]_v &= \texttt{get >>= \textbackslash } v \texttt{ -> } [\![e]\!]_v \\
&\quad \text{if } fv(e) \cap v \neq \emptyset \\
[\![e]\!]_v &= [\![e]\!]_v \\
&\quad \text{otherwise}
\end{aligned}
$$

B Well-typed processes

Typing judgments for processes and continuations are given below. $\tau \rightsquigarrow$ is the type of a continuation waiting for a reply of type τ. Reduction contexts are given function types, treating their hole as a normal, abstracted variable.

$$\frac{\Gamma \vdash K : \tau \rightsquigarrow \qquad \Gamma \vdash n : \texttt{Ref } \rho \qquad \Gamma \vdash e : \texttt{O } \rho\ \tau \qquad \Gamma \vdash c : \texttt{(Time,Time)} \qquad \Gamma \vdash m : \texttt{Msg}}{\Gamma \vdash \langle n, e, K \rangle_m^c \text{ well-typed}}\ \textsc{Pending}$$

$$\frac{\Gamma \vdash K : \tau \rightsquigarrow \qquad \Gamma \vdash s : \rho \qquad \Gamma \vdash e : \texttt{O } \rho\ \tau \qquad \Gamma \vdash c : \texttt{(Time,Time)} \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s, e, K |\!)_n^c \text{ well-typed}}\ \textsc{Active}$$

$$\frac{\Gamma \vdash m : \texttt{Msg}}{\Gamma \vdash \langle\rangle_m \text{ well-typed}}\ \textsc{Empty} \qquad \frac{\Gamma \vdash s : \rho \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s |\!)_n \text{ well-typed}}\ \textsc{Idle}$$

$$\frac{\Gamma \vdash P \text{ well-typed} \qquad \Gamma \vdash P' \text{ well-typed}}{\Gamma \vdash P \parallel P' \text{ well-typed}}\ \textsc{Parallel}$$

$$\frac{\Gamma, n : \tau \vdash P \text{ well-typed}}{\Gamma \vdash \nu n.P \text{ well-typed}}\ \textsc{Restriction}$$

$$\frac{\Gamma \vdash s : \rho \qquad \Gamma \vdash \mathcal{M} : \tau \to \texttt{O } \rho\ \sigma \qquad \Gamma \vdash K : \sigma \rightsquigarrow \qquad \Gamma \vdash n : \texttt{Ref } \rho}{\Gamma \vdash (\!| s, \mathcal{M}, K |\!)_n : \tau \rightsquigarrow}\ \textsc{Cont}$$

$$\frac{}{\Gamma \vdash 0 : \tau \rightsquigarrow}\ \textsc{EmptyCont} \qquad \frac{\Gamma, [\,] : \tau \vdash \mathcal{M}[\,] : \texttt{O } \sigma\ \rho}{\Gamma \vdash \mathcal{M} : \tau \to \texttt{O } \sigma\ \rho}\ \textsc{Context}$$

APPENDIX E

Programming with Time-Constrained Reactions. Johan Nordlander, Magnus Carlsson, and Mark Jones. Submitted for publication, April 2004.

Programming with Time-Constrained Reactions

Johan Nordlander
Luleå University of Technology
nordland@csee.ltu.se

Magnus Carlsson*
magnus@carlssonia.org

Mark P. Jones
Oregon Health & Science University
mpj@cse.ogi.edu

ABSTRACT

In this paper we argue that a programming language for real-time systems should support the declaration of *time-constraints*, and that those constraints should attach to a well-developed notion of *reactions*. To make our claims more precise, we introduce *Timber*, which is a concurrent programming language based on a model of non-blocking, reactive objects. Timber supports both upper and lower time constraints on a reaction, where an upper constraint corresponds to a classical deadline, and a lower constraint constitutes a very efficient way of scheduling an event to occur at a well-defined point in the future. A series of programming examples illustrates how these mechanisms can be used to express simple solutions to common problems in practical real-time programming.

1. INTRODUCTION

Look up just about any definition of the term *real-time system*, and it is bound to be based on some variant of the notion of "meeting a deadline". The exact wordings will of course vary, but there is little controversy in saying that in the *problem domain* of real-time systems, in particular the so called *hard* ones, the deadline concept plays a fundamental role.

It is therefore rather surprising that the real-time *solution* domain — by which we mean real-time operating systems, languages, middleware, etc — is frequently free from any references to deadlines. This is indeed a paradox, but it is a widely accepted one; its basis is a common understanding that deadline-driven approaches are yet not mature enough to be applied in anything but perhaps abstract system models. The real-time system designer is thus presented with a *semantic gap* to cross: what the problem domain refers to as events, reactions, and deadlines must be mapped onto the threads, synchronization points and static priorities that have long dominated real-time system implementation practice. It is no exaggeration to claim that bridging this gap is one of the major challenges that face anyone involved in real-time system design.

This paper takes as its starting point the observation that our common programming languages play a fundamental role in the mismatch between real-time programming practice and its problem domain. Specifically, we note that a major reason why deadlines cannot be comfortably handled in these languages is simply that the corresponding notions of events and reactions are cumbersome to express as well. The cause can be found in the I/O model that underlies modern languages, which presupposes that event handling is an internal affair of the operating system, and that the best way of interfacing an application to the real world is by means of "services" that silently block execution whenever the world is out of sync. What the programmer at best can hope for is some middleware abstraction that provides an *encoded* notion of events, and perhaps a design pattern that gives the ramifications of how reactions to events must be coded in order to fit the framework. By necessity, the possibilities of maintaining a deadline-oriented view of such a real-time system — from its specification, via an implementation, down to the actual scheduling of code at run-time — are limited.

For an analogy, consider the prospect of maintaining the type safety of function calls in a language that lacks any notion of types as well as functions! Few people would argue that such an arrangement is the best way of promoting the idea of type-correct programming. It is the core message of this paper that the issue of real-time correctness can, and should, be addressed from a programming language perspective with the same level of interest as type-correctness has received in the past.

In previous work we have described the event-driven *reactive objects* programming model that forms the basis upon which our real-time programming language Timber is built [10]. The key aspect of the reactive objects model is that events and reactions are unified with messages and methods, respectively, and that methods are non-blocking by construction. The present paper extends this work by the following contributions:

- We provide a detailed definition of and motivation for the *time-constrained reactions* that give Timber its unique character.
- The pragmatics of programming real-time systems in Timber is illustrated in a series of archetypical examples.

The perspective taken in this paper is that of the working real-time programmer with an interest in language design and programming models. The formal semantics of Timber, which is of primary interest to language implementors and

*The work was carried out while the author was at Oregon Health & Science University.

scheduling theoreticians, is defined elsewhere [3]. A third, more application-oriented view of Timber is presented in a companion paper, where the implementation of components for robot control is described [8].

The rest of this paper is organized as follows. In section 2 we review the reactive objects model of Timber, and introduce the reader to the basic structure of the language. Section 3 defines the notion of time-constraints in Timber, and shows the basic constructs by which the programmer can control the timing behavior of a Timber system. Then follows a section that exemplifies the use of time-constraints in practice (section 4). Some related approaches are reviewed in section 6, while section 7 concludes with some pointers towards future work.

2. BACKGROUND

2.1 Reactivity

The idea that the primary role of a computer in its environment is to react to events is by no means new, it has been around in various forms since the dawn of computers more than half a century ago. Constructors of embedded systems in particular have no problems embracing this view, as it clearly emphasizes the computer as a sub-component of a more diverse and general system.

The reactive view is however dwarfed by the much more persistent idea that a computer is the active part, and any equipment with which it may need to interact primarily exist as sub-components of the computing system. This view certainly holds for the large class of external devices that deal with data storage; for example, hard disks, RAM memory, card readers, and line printers. Born within the batch-oriented era of computing, the active, computer-centric perspective naturally leads to the idea of synchronization by transparently blocking input operations, dressed up as subroutines.

Of course, treating interactive users, physical processes, or the internet as mere sub-components of a particular computer is not viable, but neither were such systems commonly connected to computers when batch-oriented computing was the rule. Unfortunately, modern programming languages prevail in upholding the batch-oriented view of the world when it comes to environment interaction, something which, for example, manifests itself in the common modeling of input devices as data files.

The result is that in order to implement real event-driven applications, the programmer has to go to some length in establishing a reactive program structure out of the suggested active programming model. As an illustration, consider the typical top-level *event loop* structure that forms the core of nearly every modern interactive application. An outline of the event-loop pattern is shown in Figure 1. The goal of this pattern is to enable the core parts of an application to be written as a set of *event handlers* that are called when an event occurs. In order to achieve this, however, the calls to the blocking event delivery service of the operating system (tentatively called *GetNextEvent* in this example) must be centralized to a top-level dispatch loop, with the implicit side-condition that none of the event-handling subroutines make any call to this function, or to any indefinitely blocking

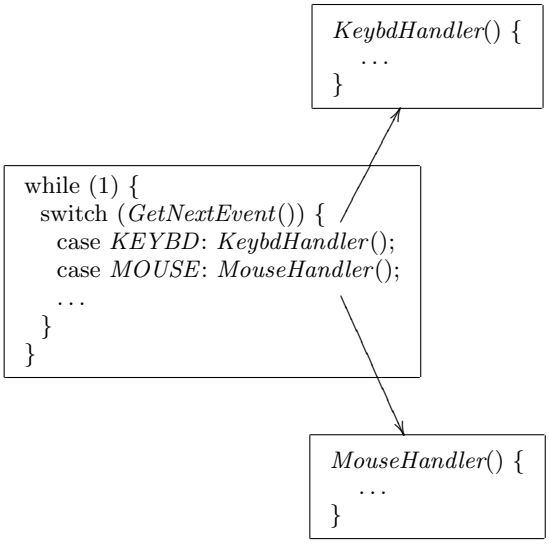


Figure 1:

service general. Thus, the role of the event-loop pattern is to convert the active *compute-until-something-happens* style of the operating system service into the passive *call-me-when-something-happens* style of the application.

Event-loops are so common in event-driven programming that many people consider the notions synonymous, despite the inherent weaknesses of the event-loop pattern: (1) operating system dependency and limited extensibility, (2) brittleness in presence of changes that break the blocking/non-blocking invariants, and (3) unclear interaction with concurrency. Moreover, the packaging of real-world events as the result of a transparently blocking operating system service is likely to be questioned by very few. Still, there is a backside to the event-loop pattern hidden inside the implementation of such an service, that has high relevance in a discussion on the structure of event-driven programs.

Consider the extended software outline shown in Figure 2. Here we have complemented the application event-loop with parts of the operating system as well, in order to illustrate the path an event takes from the initial hardware interrupt to the designated event handler. The main job of the operating system is here to use scheduling to implement blocking of the application, on basis of data generated by the low-level interrupt handlers. A key technicality is the need for some asynchronous signaling mechanism, that decouples the execution of the interrupt handlers from the rest of the system. Notice that the effort of the application to switch polarity of an active event service into a reactive program structure is mirrored within the operating system, as a mechanism for converting the reactive nature of interrupt handling into an actively blocking operation!

The irony is that so much operating system effort goes into establishing the batch-oriented ideal of transparently blocking subroutines, when the reactive programmer subsequently will have to work hard in order to predict, control and avoid this very feature. The question that has spurred the develop-

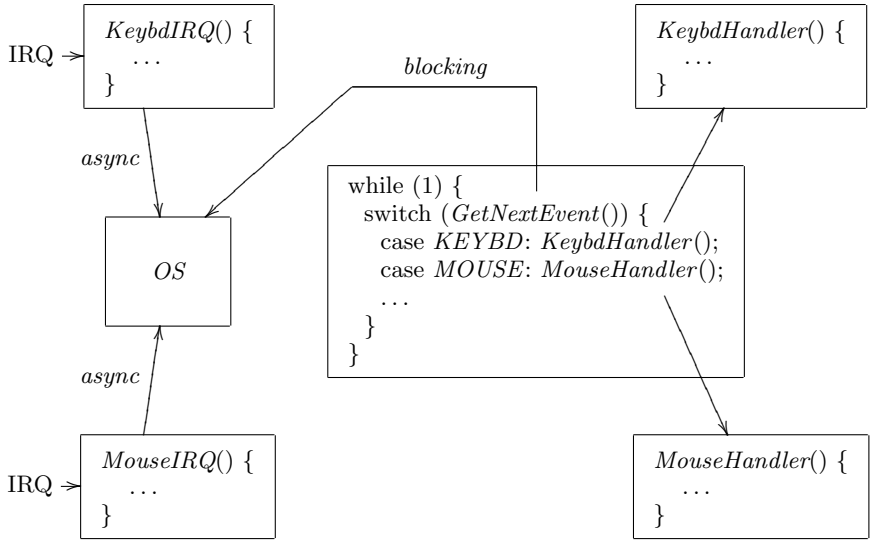


Figure 2:

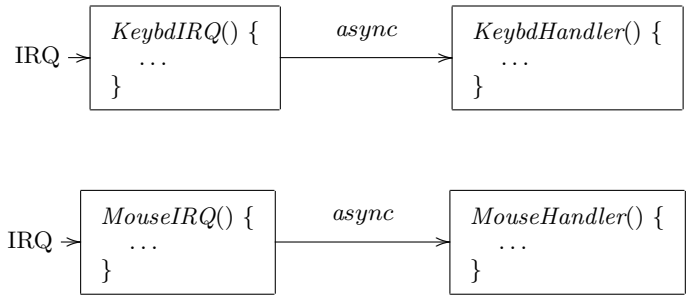


Figure 3:

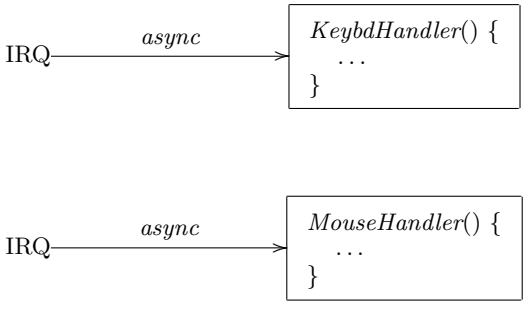


Figure 4:

ment of Timber and its programming model may seem obvious in this context: Why can't we simply bypass the double switching of polarities in traditional event-driven programming, and pipe the internal asynchronous signals of the operating system directly to the corresponding event handlers? A sketch of a reactive program structure along those lines is shown in Figure 3. Indeed, by modeling interrupt requests as asynchronous signals to begin with, one could even go one step further, as in Figure 4.

2.2 Reactive objects

Of course, the real issues in a move that bypasses the operating system must concern the semantics of the intended programming language, especially regarding concurrency and global state management. Ada and RT Java are both examples of concurrent languages with abilities to run programs on bare metal CPUs; however, both these languages presuppose that their run-time systems take over the operating system role of implementing indefinite blocking. Other, more experimental languages exist that take on a higher level view of interaction, but their programming models are often too far removed from the perspective of the embedded systems programmer (see Section **??** for some further discussion).

The Timber approach has been to leverage on the object-oriented paradigm, in order to achieve a concurrent programming model that lies very close to what is already familiar to anyone acquainted with event-driven programming. Thus, the *reactive objects* semantics of Timber is a model where

- **Each object is independent**; that is, method execution in one object is implicitly concurrent with activity in other objects.
- **State is local and protected**; that is, each piece of program state belongs to some particular object, which is the only entity that can read and modify its value.

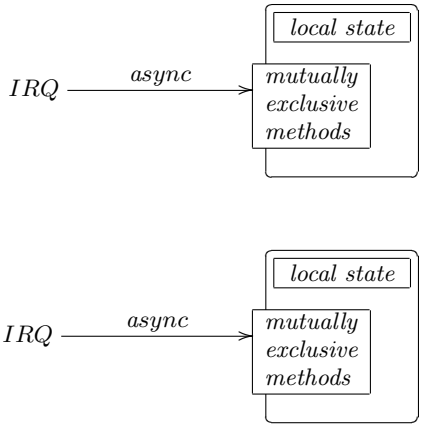


Figure 5:

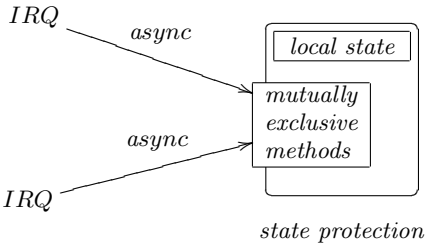


Figure 6:

Furthermore, object state cannot be concurrently accessed, as the execution of methods of a particular object is mutually exclusive.

- **Method execution is non-blocking**; that is, there is no way for a method to halt execution midway, waiting for a future event. Events are uniformly interpreted as method calls, and in the absence of diverging calculations, each method will eventually terminate and return its object to a resting state.

We illustrate the impact of these characteristics by continuing our event-driven example from above. In Figure 5 the two objects can handle their respective interrupt signals in any interleaving, as they only have access to their own private parts of the program state. The enforced non-blocking nature of methods also coincides very well with the informal requirements that usually constrain both low-level interrupt handlers and more general handlers of event-driven frameworks.

Figure 6 illustrates the case where two handlers share their state. This is expressed in the reactive objects model by letting the handler methods be part of the same object, which then guarantees that their executions will not be interleaved. The underlying asynchronous message mechanism of each object captures both any interrupt queuing facilities offered by some hardware, and the event queues normally found on a per program basis inside operating systems.

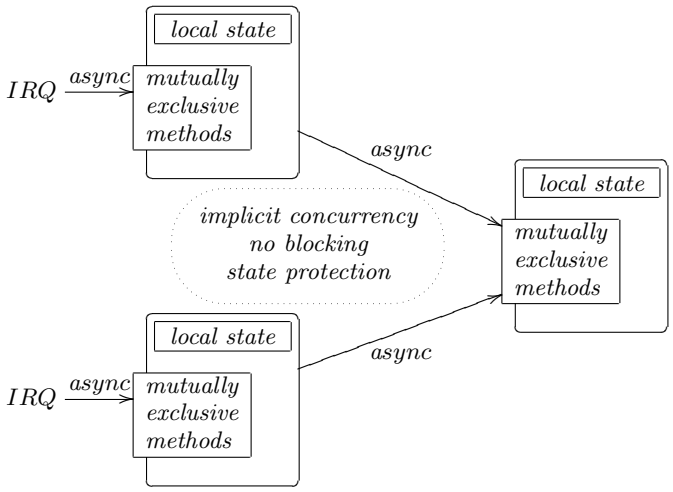


Figure 7:

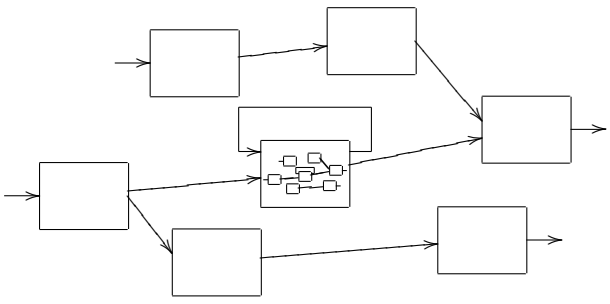


Figure 8:

In Figure 7, the possibilities of event-handling using a combination of objects is exemplified. The two leftmost objects in the figure are able to handle events concurrently, while handling of the secondary events in the right object will be serialized. What we see here is the beginning of a program structure set up to handle more complex tasks. An even more general case is shown in Figure 8, where each gray box may be a single object, or a compound structure of the same kind as the whole figure itself. The important characteristic is that the arrows represent message-based communication both in the object-oriented and in the concurrent programming sense, and that the nodes stand for a system partitioning according to state as well as control.

In general, a structure of reactive objects connects a set of input events to a set of output events, where the definition of the nodes determine how an individual input event propagates through the graph and modifies its state. At the hardware/software boundary, the input events correspond to interrupts, and output events are denoted by device register accesses. The model is however equally applicable to any level of software component abstraction, as well as to communicating systems in general, of an arbitrary scale.

2.3 Timber

With *Timber* we concretize the reactive objects model in a programming language that is further characterized by

- Synchronous as well as asynchronous communication.
- First-class methods and object generators.
- Strong type safety.
- A syntax and so called *monadic* semantics influenced by functional languages.

Here we will just briefly overview the constructs of Timber that are most relevant to the rest of the paper. For a more complete account, we refer to the draft language report [2] and the formal semantics definition [3].

On the top level, a Timber program is a set of bindings of names to expressions, of which some may be *templates*, which is the Timber term for object generators (i.e., classical *classes*). In concrete Timber syntax, a generator of simple counter objects can be defined as follows:

```
counter = template
            value := 0
            return
              { read = request
                       return value
              , incr = action
                       value := value + 1 }
```

This definition reveals that *counter* objects consist of a local state variable *value* initialized to 0, and a public interface with methods for reading and incrementing the state. The interface is a record in this case, but can in principle be any data structure. However, interfaces need to contain at least one method in order to be practically useful.

Methods are of two kinds: *actions* that imply asynchronous communication, and *requests* for expressing synchronous rendezvous. The general forms for method and template expressions are

```
action      request     template
  statements  statements  statements
```

where each statement is basically either an assignment to local state, a template instantiation, or (with the exception of template expressions) an asynchronous or synchronous method call. The usual selection of looping and branching constructs is also available, as well as a statement that introduces local bindings:

```
let
  test env = action
               c ← counter
               c.incr
               c.incr
               n ← c.read
               env.putStr (show n)
```

This example says that *test* is an asynchronous method which instantiates a *counter* object, increments it twice, reads its current value, and prints a string representation of the value to any object supplied as the parameter *env*. The ← syntax introduces a local name for the result of a side-effecting command, in contrast to the symbol =, which is reserved for declarative definitions. In this context it is worth noting that := denotes destructive update, something which is only available for state variables introduced in an enclosing template scope.

Types play an important role in Timber programming, still most type information can be left out from programs due to the automatic type inference facilities supported by the language. A cornerstone in this process is however the definition of interface signatures, which for our counter above would look as follows:

```
record Counter =
  incr :: Action
  read :: Request Int
```

From this information, the inference process will be able to deduce that *counter* has the type *Template Counter*. We will not pursue the type system aspect any further in this paper, though, and the coming examples can if necessary be read as if Timber were an untyped language.

3. TIME CONSTRAINTS

Given a programming model that identifies methods with reactions, and method calls with events, it becomes natural to associate methods with real-time constraints. Timber actually supports the declaration of both upper and lower time bounds, where an upper bound is the equivalent of a classical deadline, and a lower bound corresponds to the generation of clock-based events typically expressed using delays, timers, and timeouts in a traditional setting.

3.1 Deadlines

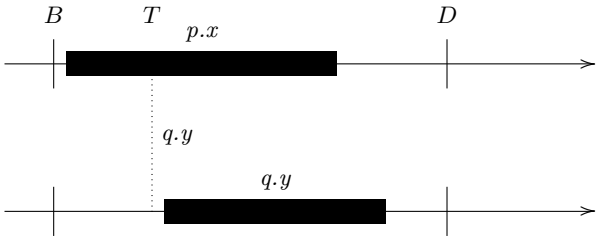


Figure 9: Inheriting time-constraints.

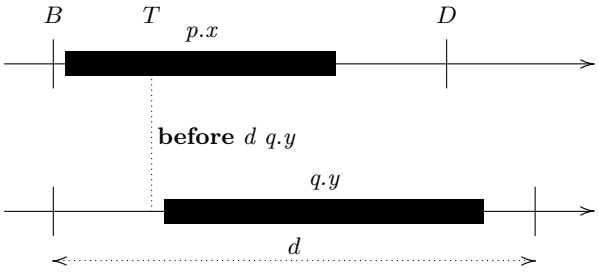


Figure 10: Extending the deadline.

The deadlines that constrain a real-time program under execution are typically derived from the desired timing behavior of the system as a whole, by decreasing end-to-end deadlines with any latencies and delays introduced by external components along the signal path. This pattern could potentially be applied for the purpose of deriving deadlines for individual software components of a reaction chain as well, but we have chosen to adapt a different strategy with Timber. Our arguments are threefold:

Firstly, the points in time when software components in a reaction chain call each other are purely artificial. They will all approach the time of the original event as CPU speed increases.

Secondly, the time when an event first enters a Timber system can readily be made available to subsequent reactions, just like any ordinary parameter. This makes it unnecessary to compensate for latencies introduced by initial segments of a reaction chain.

Thirdly, if a Timber system is at all schedulable, it is implied that subsequent components in a reaction chain will also meet their deadlines. Compensating for latencies in these components will thus not improve the schedulability of the system.

For these reasons, the model used in Timber is that a deadline by default applies to *all* reactions that might result from an event entering the system. Internal events are thus no more observable than the individual statements executed by reactions; all that matters is that every component involved in the handling of a particular event is able to complete before the given end-to-end deadline, measured from a common reference point in time.

We illustrate this idea with a timeline of two coupled method executions in Figure 9. Here an event enters the system at time B as an interrupt coupled to some method $p.x$. The reaction to this event must be finished before time D, which of course implies that $p.x$ must complete before D. However, $p.x$ also makes an asynchronous call to a method $q.y$ at some point T during its execution. Now, instead of making this point the start of a new timeframe with its own deadline—which would inescapably lead to the question of what the deadline is for reaching point T—Timber simply lets $q.y$ inherit both D as well as its reference point B. Notice that if any of $p.x$ or $q.y$ misses the common deadline under these circumstances, the system must be overloaded. In particular, specifying an artificial deadline for T would not have a chance of rectifying the problem, as the total amount of processor work required by the methods would still remain the same.

There are however situations where only *part* of a reaction is associated with a tight deadline, but other parts have much looser constraints. This need is addressed in Timber by a means for extending the deadline at each asynchronous method call. An example of such a scenario is shown in Figure 10. By prefixing the call to $q.y$ with the construct **before** d, the deadline of $q.y$ is adjusted to d, measured relative the common reference point B (note: not the time of the call). The new effective deadline is however dynamically limited to values greater than or equal to the current deadline, in order to avoid paradoxical situations where a secondary reaction ends up more urgent than the activity that has caused it! The ability to extend deadlines is furthermore only available for asynchronous method calls. A synchronous call, which by definition occurs interjected within activities by the caller, can neither be more urgent than the code preceding the call, nor less urgent than the code that follows.

The extension mechanism is also used to set the initial deadlines associated with external events—conceptually, the default relative deadline for such events is 0, so any positive value constitutes a legal extension. The actual representation of relative deadlines is abstract, but a variety of constant symbols is available to allow time to be expressed in most commonly used units. A particular value sets the effective deadline to infinity, thus removing the reaction in question from the real-time domain. It must be stressed, though, that while the Timber scheduler bases its run-time decisions on the deadlines attached to each reaction, no attempts are made to automatically detect and handle deadline misses. The correct behavior in such situations is highly application-dependent, and we will see in the following sections how detection of missed deadlines can be easily programmed. Furthermore, we believe that the greatest value of the Timber deadline regime lies in its semantic formalization of what constitutes time-correct program execution [3]. This is a necessary starting point for any form of static schedulability analysis, something which is of undisputed importance to the field of hard real-time systems. We are currently in the process of developing the basis of such an analysis for Timber.

3.2 Baselines

As a dual to the upper, "must-finish-before" time-bounds we refer to as deadlines, we introduce the term *baseline* for the common reference point of an event we have been introduced

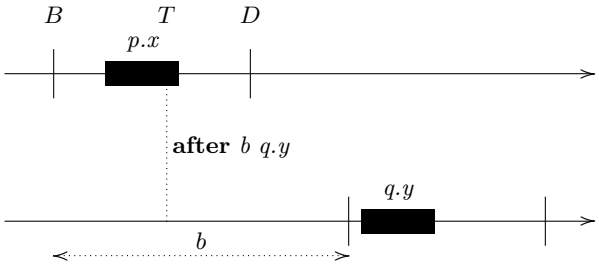


Figure 11: Extending the baseline.

in the previous subsection. A baseline can be thought of as a "must-start-after" constraint, even though it is obvious that as long as baselines are solely identified with time-stamps of external events, such constraints will always be met.

However, just as it is meaningful to extend the deadline of selected method calls, so is the concept of increasing baselines. Analogous to the **before** construct, we may write **after** b in front of an asynchronous call to shift its baseline b time units into the future, again counting from the baseline of the caller. An illustration of a baseline shift is shown in Figure 11. Notice especially that in the absence of an explicit deadline prefix, the deadline of a call remains constant *relative its effective baseline.*

By means of the **after** construct, the Timber programmer is given the opportunity to schedule asynchronous method calls at well-defined points in the future, that are insensitive to variations in execution speed of the current statement sequence. Moreover, when combined with the ability to control deadlines, the window for executing these methods can be specified with arbitrary precision. A particular advantage with this mechanism is the maintained ability to react while calls with extended baselines are in transit. In line with the reactive object model, there is no notion of sleeping or blocking involved, neither by the caller, nor by the callee. We will see in the following section how this feature may be put to good use.

4. USING TIME-CONSTRAINTS

In this section we will illustrate the practical side of programming with time-constrained reactions by means of a series of program fragments.

4.1 Periodic tasks

Our first example will be a pattern for constructing a periodic task, by utilizing a recursive asynchronous method with a shifted baseline:

```
let tick = action
             stmts
             after (50 * ms) tick
```

This method can be included in any kind of template context; other methods of the same object will peacefully coexist with the periodically activated code. Because the baseline offset $50 * ms$ is measured relative the current baseline at each invocation (and not the actual time of the call), the implementation does not suffer from accumulating drift. Notice that an initial call to *tick* is required to start the process, so in a sense, the periodic execution that follows can be seen as an infinitely oscillating reaction to that first triggering event.

As an illustration to the usefulness of explicit deadlines, we can easily define a variant of the code above where the time slot available for *stmts* is much less than the period:

```
let tick = action
             stmts
             after (50 * ms)
               before (7 * ms) tick
```

If we want to specify that *all* calls to *tick* (including the first one) should have the same relative deadline, we might write as follows instead:

```
let tick = before (7 * ms)
           action
             stmts
             after (50 * ms) tick
```

Notice how the **before** prefix can be applied to both the name of a method at a call site, and its actual definition. With the referentially transparent semantics of Timber, these alternatives both amount to the same thing!

4.2 Implementing delays

A non-repetitive call with an extended baseline is shown in the following example, which implements a simple delay between commands to a slow external device like a plotter.

```
let move pos = action
                 pen.ctrl Up
                 xy.moveto pos
                 after (200 * ms)
                   (action pen.ctrl Down)
```

As always with the **after** construct, the invoking object is free continue immediately after the invocation. The example also illustrates the use of an *anonymous* method; i.e., a method not bound to a name.

The case where data must be read at the end of a specified interval must be coded in Timber by means of a *callback* method; i.e., a parameter standing for an unknown method accepting the desired data.

```
let readAt pos cont = action
                        head.moveto pos
                        after (200 * ms)
                          ( action
                              data ← head.read
                              cont data )
```

A synchronous method that simply returns the data after 200 ms cannot be written in Timber, as there exists no means of blocking execution for that long while a method is active. The minor awkwardness of having to supply a callback parameter should however be contrasted with the maintained responsivity of both the caller and the callee in the Timber formulation. The readiness to respond to new

events during the 200 ms time-slot should have direct consequences for the ease by which (say) monitoring and error handing could be added to the template fragment above.

4.3 Timeouts

In general, a Timber object will keep track of what it has done and what to expect by means of local state variables. As an illustration, the following code example implements parts of an object that issues requests over a network, after which it expects replies within a specified interval.

```
template
  state := Idle
  let
    send    = action
                network.sendreq reply
                state := Active
                after t timeout
    reply d = action
                if state == Active then
                  ...
                  state := Idle
    timeout = action
                if state == Active then
...               ...
```

In the call to *network.sendreq* we see a typical example of how a callback parameter is supplied to handle a potential reply. Notice how the two different outcomes—a real reply, or a timeout—are given equal status in the implementation. One may also remark that since the timeout value is measured relative to the baseline for the *send* call, the timeout counter will start counting slightly before the request packet appears on the network. If it is important that this deviation is controlled and limited, the programmer has the option of specifying an explicit deadline for *send*.

In the previous example, it is manifest that both a reply and a timeout might occur while the sending object is in state *Idle*. The fact that an external network node is fully able to send anything it wants at any time cannot be ignored, so the state check performed in *reply* must really be present in any robust implementation. The timeout events, on the other hand, are actually under local node control, so it might seem reasonable to have them cancelled whenever a valid reply is received.

Timber offers such an opportunity be means of unique *message tags* that are created as the result of each asynchronous call. Normally these tags are just ignored, but in case the ability to cancel a specific call is needed, its tag should be captured and stored. Cancellation amounts to calling the special *cancel* method of the designated tag, as the following fragment shows:

```
tag ← after t timeout
...
tag.cancel
```

The Timber library contains a useful template called *singlecall*, which takes care of managing the storage of a single tag, as well as automatically canceling any old message whenever a new supervised call is requested. A reformulation of the network request example illustrates the use of *singlecall* to cancel redundant timeouts.

```
template
  state := Idle
  sc ← singlecall
  let
    send    = action
                network.sendreq reply
                state := Active
                sc.call (after t timeout)
    reply d = action
                if state == Active then
                  ...
                  sc.cancel
                  state := Idle
...timeout = action ...
```

4.4 Missing deadlines

As deadline misses are not automatically caught by Timber, such detections must be programmed explicitly. Due to the consistent use of baselines for timing reference, it is easy enough to define a task that checks some application-dependent program state when the deadline for some computation is due:

```
before t action
        after t other.checkResult
        ... do heavy computation ...
        other.deliverResult
```

There are however many design decisions to make before this pattern can be turned into concrete code. Should deadline misses be handled exactly when they occur, or is it ok to wait until the over-running computation is done? Should the failing computation be aborted, or is it sufficient to simply replace or disable the deliverance of its result? What should replace the result if the computation is aborted, and in what state should that object be left? And how do we deal with the fact that deadline-based systems in general are inherently unpredictable at overload situations?

Timber does provide an experimental feature for aborting running computations (in contrast to canceling pending messages), but we have not reached any conclusion whether this is the right level for handling missed deadlines. While it is clear that the need for aborting running computations is much less pressing in Timber than in languages based on blocking threads, we wish to point out that a thorough study on overload handling in Timber must remain as a topic for future work.

4.5 A time-constrained data collector

As an illustration of a reaction chain where deadlines are gradually relaxed, consider the following sketch of the data collecting part of a program for data acquisition and analysis.

collector analyzer

```
= template
    buf := []
    let
      irq  = before (4 * microseconds)
             action
               data ← inport.read
               buf := (baseline, data) : buf
      t    = 2 * ms
      tick = before (100 * microseconds)
             action
               before t (analyzer.analyze buf)
               buf := []
               after t tick
    return { irq = irq; start = tick }
```

```
template
  t0 := genesis
  let
    ping = before (50 * microseconds)
           action
             port.write beepOn
             t0 := baseline
             after (2 * ms) stop
             after (2 * seconds) ping
    stop = action
             port.write beepOff
    echo = before (5 * ms)
           action
             if baseline − t0 < limit then
               alarm
  return { irq = echo; start = ping }
```

Here the intended scenario is that the availability of new data is announced through an interrupt, after which a data value must be fetched from a specified register within 4 μs. Moreover, every 2 ms the collected data is handed over to an analyzer for further processing. The deadline for analyzing data is a full cycle of 2 ms; however, the jitter for these cycles must not exceed 100 μs.

These timing requirements are expressed in Timber as a deadline of 4 μs for the interrupt handler, and a deadline of 100 μs for the periodic tick that drives the analyzer. However, since the analyzer itself may take up to 2 ms to complete its task, the hand-over call is assigned a relaxed deadline by means of the **before** construct.

In this example it is especially important to recall that explicit deadlines in Timber code denote *specifications*, not statements about expected execution times. Since the constrained methods *irq* and *tick* are part of the same object, they are bound to occasionally obstruct each other. That impact is however limited to the *execution times* of each method, and it is the task of a schedulability analysis to figure out if these times are short enough to allow both methods to always meet their deadlines.

4.6 A sonar driver

We will end this exposition of time-constrained programming in Timber by showing a simple implementation of a sonar driver that is coupled to an alarm. The specifications we are assuming state that a sonar beep should be 2 ms long, with a maximum jitter of 50 μs, and that the required accuracy of the measurements dictate that time-stamps associated with beeps must also be accurate down to the 50 μs range. Furthermore, the sonar is supposed to sound every 2 seconds, and the deadline for reacting to off-limit measurements is 5 ms. These specifications look as follows when translated into Timber code:

```
sonar port alarm =
```

5. RELATED WORK

The complexity of event-driven programming in mainstream languages is illustrated by the existence of formalized real-time models like ROOM and Rational Rose Real-time [13, 4]. The purpose of these models is to automatize as much as possible of the often tedious and error-prone construction of an event-driven infrastructure, by taking abstract system descriptions in terms of actors, messages, etc; and generating the major part of a corresponding implementation in a language like C++ or Java, mapped onto some specific operating system. However, despite references to the term real-time in their names, neither of these models support an especially well developed notion of time. This fact is addressed by Saksena *et al*, who extend ROOM with the concept of deadlines, and provide a set of guidelines for mapping these problem domain notions onto the priority-based thread abstractions of the presumed target system [12]. While these guidelines undeniably identify important areas of real-time system design, they can also be seen as a very concrete symptom of the semantic gap that does exist between the problem and solution domains of real-time systems.

In the realm of more domain-specific, but also more experimental, real-time programming languages we find designs that do provide general forms of timing constraints that, at least in principle, could open up for deadline-based scheduling. Examples are Real-time Euclid [9], RTC++ [16], and CRL [14]. Hooman and van Roosmalen describe a generic language extension in the same spirit, exemplified with an unnamed language design that even comes with a formal definition [6]. However, a general remark regarding these approaches is that timing constraints apply to very fine grain program units (statement blocks in RTC++, individual statements in CRL and the Hooman/van Roosmalen design); which, by the presence of general threads and blocking constructs, do not correspond very well to the actual schedulable units as they appear at run-time. It is also the case that timing constraint arithmetic on this detailed level risks being rather elaborate, something which further reduces the value of these languages as real-time modeling tools.

Real-time Euclid is moreover an example of a language where resource-awareness has motivated severe restrictions in expressivity in order to simplify execution time and schedulability analysis [9]. This sparseness is also shared by some

real-time languages based on the functional programming paradigm [15, 7, 11]. Here, however, the term real-time is actually identified with bounds on resources, not with any means for declaring constraints that specify some *desired* timing behavior.

Languages for reactive programming are often associated with the *synchronous* model of computation (Esterel, Signal, Lustre) [1]. In synchronous languages, computations are assumed to take place at specific instants, rather than being spread out in time. In practice this is achieved by conducting program execution by one or more periodic clocks, where all computations are assumed to terminate within a clock period. The need for scheduling and schedulability analysis is thus removed, at the expense of a fully synchronized system whose responsiveness is determined by the longest running computation. In contrast, each reaction in a Timber system can be individually constrained, and arbitrary approximations to the idea of instantaneous computation can be achieved by placing upper and lower time bounds sufficiently close.

Another interesting variant of the fully time-driven approach is the language Giotto and its underlying model in the shape of the E-machine [5]. Here the programmer can specify that a reaction to an event should be delivered precisely at a certain point in time. Consequently, the output of a task will be queued until it is time to react, and so the E machine becomes a deterministic system. We take it as an axiom that a fully time-driven and deterministic approach constitutes an over-specification when used for the purpose of modeling general time-constrained systems.

6. CONCLUSION AND FUTURE WORK

In this paper we have argued that a programming language for real-time systems should support the declaration of *time-constraints*, and that those constraints should attach to a well-developed notion of *reactions*. To make our claims more precise, we have introduced *Timber*, which is a concurrent programming language based on a model of non-blocking, reactive objects. Timber supports both upper and lower time constraints on a reaction, where an upper constraint corresponds to a classical deadline, and a lower constraint constitutes a very efficient way of scheduling an event to occur at a well-defined point in the future. A series of programming examples has illustrated how these mechanisms can be used to express simple solutions to common problems in practical real-time programming.

Several directions for future work exist. One obvious path concerns finalizing a public Timber implementation, in order to enable gathering of experiences from a more diverse field of programmers. We have also mentioned that the overload behavior of Timber, not to mention deadline-based systems in general, is an area that requires further study.

In terms of program analysis for Timber, we would like to attack the general schedulability problem as well as the problem of establishing method execution times on various platforms. For both problems, we expect Timber to offer some interesting opportunities; the fact that state variables are guaranteed to be free from concurrency interference should for example simplify the task of establishing flow information.

Finally, even though the deadline-based semantics of Timber lies very close to the established field of EDF scheduling theory, there does not seem to be a straight-forward way of determining resource requirements for the scheduled chain reactions made possible by our **after** construct in any existing analysis framework. This is an interesting new research problem that we are looking forward to study in more detail.

Acknowledgments

The work reported in this paper was sponsored in part by DARPA, contract #F33615-00-C-3042, as part of the PCES program (Program Composition for Embedded Systems). This work has benefited from the comments of members of the Project Timber team and of the PacSoft center at OGI. Additional thanks to Jan Jonsson and Björn von Sydow for insights and helpful discussions.

7. REFERENCES

[1] A Benveniste, P Caspi, S A Edwards, N Halbwachs, P Le Guernic, and R de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[2] A.P. Black, M. Carlsson, M.P. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems. Technical Report CSE-02-002, Dept. of Computer Science & Engineering, Oregon Health & Science University, April 2002.

[3] Magnus Carlsson, Johan Nordlander, and Dick Kieburtz. The semantic layers of Timber. In Atsushi Ohori, editor, *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China*, volume 2895 of *Lecture Notes in Computer Science*. Springer, November 2003.

[4] G. Gullekson. Designing for concurrency and distribution with rational rose realtime. White paper, Rational Software, 2000.

[5] Thomas A. Henzinger and Christoph M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[6] J. Hooman and O. van Roosmalen. An Approach to Platform Independent Real-Time Programming. *Real-Time Systems, Journal of Time-Critical Computing Systems*, 19(1):61–112, 2000.

[7] John Hughes and Lars Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[8] Mark Jones, Magnus Carlsson, and Johan Nordlander. Composed, and in control: Programming the Timber robot. `http://www.cse.ogi.edu/~mpj/timbot/ComposedAndInControl.pdf`, 2002.

[9] E. Kligerman and A.D. Stoyenko. Real-Time Euclid: A Language for Reliable Real-ime Systems. *IEEE Transactions on Software Engineering*, SE-12(9), 1986.

[10] Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, April 2002.

[11] Alvaro Rebon Portillo, Kevin Hammond, Hans-Wolfgang Loidl, and Pedro Vasconcelos. Granularity analysis using automatic size and time cost inference. In *Proceedings of IFL '02—Implementation of Functional Languages*. Springer Verlag, September 2002.

[12] M. Saksena, P. Freedman, and P. Rodziewicz. Guidelines for Automated Implementation of Executable Object Oriented Models for Real-Time Embedded Control Systems. In *IEEE Real-Time Systems Symposium*, 1997.

[13] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.

[14] A.D. Stoyenko, T.J. Marlowe, and M.F. Younis. A Language for Complex Real-Time Systems. *The Computer Journal*, 38(4), 1995.

[15] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *International Conference on Functional Programming (ICFP '01)*, Florence, Italy, September 2001.

[16] Y. Ishikawa et al. Object-Oriented Real-Time Language Design: Constructs for Timing Constraints. *SIGPLAN Notices*, 25(10):289–298, Oct 1990.

APPENDIX F

Composed, and in Control: Programming the Timber Robot. Mark Jones, Magnus Carlsson, and Johan Nordlander, Technical Report, August, 2002.

Composed, and in Control: Programming the Timber Robot

Mark P. Jones[1], Magnus Carlsson[1], and Johan Nordlander[2]

[1] Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU, Beaverton, OR 97006, USA
[2] Department of Computing Science
Chalmers University of Technology, S - 412 96, Göteborg, Sweden
http://www.cse.ogi.edu/pacsoft/projects/Timber/

Abstract. This paper describes the implementation of control algorithms for a mobile robot vehicle using the programming language Timber, which offers a high-level, declarative approach to key aspects of embedded systems development such as real-time control, event handling, and concurrency. In particular, we show how Timber supports an elegant, compositional approach to program construction and reuse—from smaller control components to more complex, higher-level control applications—without exposing programmers to the subtle and error-prone world of explicit concurrency, scheduling, and synchronization.

1 Introduction

This paper describes some programs that we have written to control a small robot vehicle called *Timbot* (the "Timber Robot"). The functionality provided by these programs and the specifications of the vehicle on which they run are not particularly unusual from the perspective of previous and current work on autonomous control and robotics. But the novelty of our work, and the focus of this paper, lies in our use of *Timber*, a new programming language that has been designed to facilitate the construction and analysis of software for embedded systems. In particular, Timber offers a high-level, declarative approach to several of the key areas for embedded systems development—such as real-time control, event handling, and concurrency. By comparison, the languages that have traditionally been used in this domain typically relegate such features to the use of primitive APIs or coding idioms, leading to programs that are harder to analyze, and more difficult to reuse or port to new platforms.

1.1 Background: Project Timber

The work reported here has been carried out as part of a larger effort called Project Timber ("*Time* as a *b*asis for *e*mbedded *r*eal-time systems") at the OGI School of Science & Engineering. One of the goals of Project Timber, and of

particular relevance in this paper, is to investigate the role that advanced programming languages can play in supporting the construction and analysis of high-assurance, portable real-time systems.

Another important goal for the project is to develop techniques and mechanisms for building systems that adapt dynamically to changes in their environment, in resource allocations, or computational load. By comparison, many systems today are brittle, and become unusable, or perhaps even fail outright, if used under conditions that their designers had not anticipated. For example, a live MPEG video stream can quickly become unwatchable if even just a small percentage of the underlying network packets are lost as a result of a drop in available bandwidth. An alternative is to build more flexible systems that can *gracefully degrade* the quality of the services they provide—according to user-specified policies—and still provide useful functionality. In the case of a live video stream, for example, we can respond to a reduction in bandwidth by reducing the frame rate, the image size, the color depth, or some user specified combination of these, and still continue to enjoy a live, real-time video stream. In fact, this very example was one of the motivations for including a video camera on Timbot, and we are investigating a somewhat different application for Timber as a tool for programming and analyzing adaptation policies. Further discussion of these topics, however, is beyond the scope of this paper.

1.2 Outline of Paper

The rest of this paper are as follows. In Section 2, we give an overview of Timbot, describing the hardware components from which it has been assembled.In Section 3, we introduce the Timber programming language by showing how it can be used to provide a software interface to the robot vehicle. In Section 4, we begin to use this interface to develop a library of reusable *controllers*, and then we show how these can be combined to implement more complex control applications in Section 5. Finally, in Section 6, we conclude with a review of future and related work.

2 Introducing Timbot

In this section, we provide an overview of Timbot, the small robot vehicle shown in Figure 1. Built on the chassis of a radio-controlled monster truck, Timbot hosts an on-board embedded PC (an 850MHz PIII, with 256MB ram, and a wireless 802.11b network adapter); an analog video camera on a pan-tilt mounting, which connects to the computer via a PC/104+ frame grabber; sonar and line tracking sensors; and a battery system that allows Timbot to be used either on the desktop or as a standalone vehicle. Recently, we have been using Timbot with a standard Linux distribution installed on a 1GB microdrive, which provides enough headroom to host a fairly rich development environment. However, we have also used Timbot in other configurations, replacing the microdrive with smaller compact flash cards, and using custom built Linux kernels with RTAI

Fig. 1. Timbot, the Timber Robot

for real-time support. Timbot has more memory and more computational power than the machines found in many industrial embedded systems; this reflects its intended use as a platform for experimentation and demonstration. Nevertheless, it still exhibits many of the characteristics—and raises many of the challenges—that occur in the development of a modern, sophisticated embedded system.

The block diagram in Figure 2 shows the connections between the main components of Timbot in more detail. At the center, the CPU is connected over

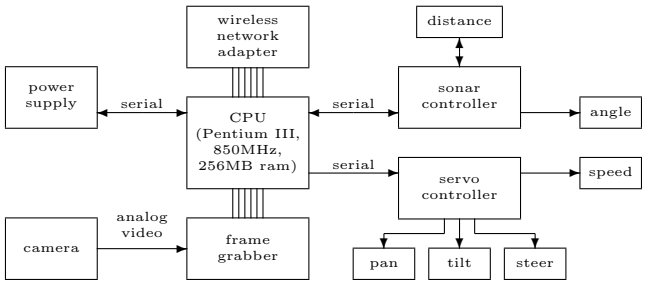


Fig. 2. A Block Diagram of Timbot

standard PC buses to a network adapter and to the frame grabber/camera combination in the lower left hand corner. Toward the top left, a power supply unit generates regulated voltage supplies for each component and handles charging of the batteries when Timbot is connected to a desktop power supply. The power supply is also connected to the CPU by an RS232 serial line that can be used, among other things, to query for an estimate of remaining battery power.

For this paper, however, our attention will be focused on the components in the right hand portion of the diagram, centered around the servo and sonar controllers. These two devices are also accessed via RS232 serial connections, and respond to multi-byte messages from the CPU by setting the physical position of a servo and, in the case of the sonar controller, pulsing an ultrasonic range finder to measure the distance to an obstacle. There are four servos on Timbot, and these can be set independently to adjust camera direction (pan), camera

attitude (tilt), steering direction (steer), and sonar direction (angle). In addition, an electronic speed controller (speed), which has the same electrical interface as a mechanical servo, is used to control the speed of Timbot's motors, and hence the speed at which it crosses the floor.

3 A Timber interface for Timbot

In this section, we describe the interface that Timber programs use to access and control the robot. This interface provides a bridge from the description of hardware in the previous section to the control programs that will come later on. We will also use this to introduce the Timber language and the idioms of reactive programming that it adopts. In the tradition of declarative languages, the goal of Timber is to allow programmers to describe *what* they want to accomplish, without having to worry about *how* it is achieved in terms of low-level concepts such as concurrent threads, scheduling, interrupts, and synchronization.

In discussing Timber code, we will comment on important details, but we do not attempt to provide a full tutorial or reference; further details may be found elsewhere [1]. In fact, Timber was derived from Nordlander's O'Haskell [3], which was, in turn, based on the functional language Haskell [7]. Where O'Haskell modified Haskell by providing a high-level approach to concurrency and reactivity via stateful objects, asynchronous messaging, and subtyping, the Timber language modifies O'Haskell by adding constructs to specify timing behavior and by adopting strict evaluation to facilitate analysis of worst-case execution times. We hope that the key aspects of our code will be clear from the accompanying text, but experience with Timber, O'Haskell, Haskell, or similar languages (in decreasing order of applicability) will be needed to understand the details.

3.1 The Timbot Interface

Our first fragment of Timber code is the definition of the `Timbot` type, which describes the software interface to the special hardware features of the robot:

```
record Timbot < Truck, CameraControl, Sonar

record Truck =
  speed :: Speed -> Action
  steer :: Angle -> Action

record CameraControl =
  pan   :: Angle -> Action
  tilt  :: Angle -> Action

record Sonar =
  angle :: Angle -> SonarListener -> Action
```

The first line tells us that `Timbot` is a combination of three interfaces for controlling vehicle movement (`Truck`), camera orientation (`CameraControl`), and

sonar usage (`Sonar`), respectively. The `Truck` and `CameraControl` interfaces are straightforward, with methods that take either a single angle or speed parameter and return an action that will move the corresponding servo to the specified position. More generally, actions represent "asynchronous message sends," and are an implicit trigger for concurrent execution, allowing new tasks to be executed (or, at least, scheduled for later execution) without delaying further execution of the code from which the action was invoked. (Note that there is no need to wait for the result of an action because actions do not return values.)

The definition of the `Sonar` interface requires more explanation. In a traditional programming language we might expect to access the sonar by calling a function that: sets the sensor direction; pulses the ultrasonic transducer; listens for an echo to obtain an estimate of distance; and returns the result to the caller. It has long been recognized, however, that so-called *blocking* operations like this are a significant source of complexity in the coding of concurrent and distributed systems [4], often requiring programmers to make assumptions, or to use encodings that can lead to deep but subtle bugs—such as deadlock—if they are not correct or if they are not applied correctly. Timber avoids these problems by eliminating blocking computations. There are no blocking primitives in the Timber library—no `getchar`, `read`, or `delay` methods, for example—nor any means for a programmer to construct a blocking method. Instead, Timber adopts a purely event-driven approach in which a program advertises methods (or "*callbacks*") that its environment can invoke to inform the program when input becomes available. Of course, this event-oriented approach is widely used in other languages, but it is weakened in many cases by the continuing presence of blocking operations, which makes it much harder to use in a reliable fashion.

For our interface to the Timbot's sonar, we avoid the need for blocking by passing two parameters to the `angle` method. The first is the angle for the sonar, while the second is a "listener" object that determines how the resulting distance reading will be passed back to the program.

```
record SonarListener =
  distance :: Maybe Distance -> Action
```

Notice that the `distance` method takes an argument of type `Maybe Distance`, indicating that it will either be a value of the form `Just x`—when a distance `x` is measured—or a value `Nothing`—when no measurement is obtained. This could occur if there is no object within range (approximately 2.7m) or if the ultrasonic signal from the sonar is absorbed instead of being reflected. By distinguishing this case explicitly within the type system, we may incur a little extra work in decoding and applying distance readings. Nevertheless, this is clearly preferable, at least if one is interested in reliable control programs, to encoding an out-of-range reading as a particular distance value, and hoping that programmers will always remember to check for the special case.

3.2 Implementations of the Timbot Interface

Every top-level Timber program is parameterized by an `env` argument of type `StdEnv` that allows the program to interact with the external environment in

which it is running. For example, the environment provides a `putStrLn` method that can be used in a command of the form `env.putStrLn msg` to display a message on the console. In the future, we plan to extend the environment with a `timbot` method that allows the interface to Timbot's hardware to be accessed in a similar way as `env.timbot`. Our current prototype, however, uses the following `getTimbot` function instead, whose implementation is discussed in Section 3.4.

```
getTimbot :: StdEnv -> Cmd Timbot
```

Of course, it is useful (and possible) to have other implementations of the `Timbot` interface. For example, we use one such interface for simple development and testing on machines other than Timbot. We are also constructing a more sophisticated implementation of the interface that connects Timber control programs to a simulator that can accurately model the motion and sensors of Timbot.

Now we can begin to write simple programs to control Timbot! The following program, for example, uses `getTimbot` to obtain an interface to Timbot (binding the variable `timbot` to the result), and then starts the robot moving at a constant speed with the steering turned all the way to the left. The result, of course, is to drive Timbot anticlockwise around the perimeter of a circle.

```
circle env = do timbot <- getTimbot env
                timbot.steer hardLeft
                timbot.speed 30
                after (20*seconds) (timbot.speed 0)
```

The `do` keyword introduces a sequence of four commands. The symbolic constant `hardLeft` gives the maximum angle to which the steering can be set for a left turn. (There is, of course, a corresponding value, `hardRight`, for right turns.) The speed setting of `30` corresponds to a (slow) forward speed; speed values range between -128 (reverse, at speed!) and +127 (forward, with haste!), but we have not yet attempted to calibrate speed in more traditional units.

Given only the first three lines, the definition of `circle` would, quite literally, send Timbot into an infinite loop: once a setting is made, the servo controller will work, even against physical pressure, to maintain it. To prevent the infinite loop, we included the final line to specify that `timbot.speed 0` should be executed 20 seconds after the program begins. The symbolic constant `seconds` is a multiplier that can be used to express time values. (There are similar multipliers for `milliseconds` and `microseconds`.) The use of symbolic constants makes it possible to express times in a platform independent manner, recognizing that multiplier values are likely to vary from one machine to the next.

This is also our first example of a timing annotation. In code like this, the `after` construct is intuitive and simple, but we will see that it is also powerful. As a first hint, we note that the `after` construct in this example is emphatically *not* the same as a 20 second (blocking) delay followed by the `timbot.speed 0` command. (Remember: there are no blocking operations in Timber!) Instead, it should be read as a high-level declaration of the time at which a specific action is to be performed. It would make no difference if the `after` construct were moved to the first line after the call to `getTimbot`; the semantics, and for all practical purposes, the observational behavior of the program would not be changed.

3.3 Monadic Programming in Timber

While Timber adopts a strict evaluation strategy like ML [2], it also relies on monadic programming [8] to encapsulate side-effects, and to facilitate analysis and optimization. Many of the programs in this paper run in the `Cmd` monad, meaning that they have a type of the form `Cmd t`, and correspond to a command that can be executed to obtain a result of type `t`. When no particular return value is needed, we typically substitute the unit type, written `()`, for `t`. For example, the `circle` program in the previous section is a function of type `StdEnv -> Cmd ()`.

In practice, most Timber programs use several different but related monads that distinguish, for example, between *methods*—which have access to the local state of an object—and *commands* (i.e., values with types of the form `Cmd t`)—which can invoke the methods of an object, but do not themselves have any local state. The use of different monads provides documentation and more precise information for programmers and program analysis tools alike. For example, the type system allows us to distinguish several special types of command using subtypes of `Cmd t`: commands that execute a synchronous request have a type of the form `Request t`; actions—which are commands that execute an asynchronous method call—have type `Action`; and commands that instantiate a template to construct an object of type `t` have types of the form `Template t`.

Timber also provides special syntax for these different kinds of command. Requests are written as a sequence of commands prefixed by a `request` keyword instead of the `do` that we saw in the definition of `circle`. Return values are specified by commands of the form `return e`. Actions are written in a similar way, but prefixed by the `action` keyword. Of course, actions do not (and cannot) specify return values. The syntax for templates uses expressions of the form `template` *local* `in` *e* to denote a command that, when executed, will construct a new object whose local state variables (if any) are initialized by the (possibly) empty list of statements in *local*, and whose interface is specified by the expression *e*. The interface to an object will often be a record, but this is not required; unlike other object-oriented languages, Timber treats objects and records as orthogonal language features.

3.4 Implementation Details

It is quite easy to implement the `getTimbot` function of Section 3.2 using standard Timber libraries that work with character devices. For reasons of space, we do not include the code here, and restrict ourselves to a brief discussion of the most important issue: synchronization. For example, from the description of Timbot in Section 2, it is clear that truck control settings and camera control settings must be multiplexed through the same serial link to the servo controller. Some kind of synchronization is needed in situations like this to avoid giving concurrent tasks simultaneous access to the same device. The sonar controller might easily become confused, for example, if the multi-byte messages for two different control tasks were accidentally interleaved. Many languages, however, require

explicit coding of synchronization, which works against the goals of abstraction because it assumes that programmers will have enough information about the underlying implementation to understand when synchronization is required and to know how it should be achieved. In Timber, these problems are solved at the language level—its semantics guarantee that each object is treated as an implicit critical section, meaning that at most one of its methods is active at a time. As such, synchronization is implicit in Timber code, with the task of determining where it is actually necessary being left to the underlying implementation.

4 The Controller Abstraction

While embedded systems may use sophisticated sensors and actuators to engage in complex interactions with their environment, many present a much simpler interface to their human users: an on/off switch! This is typical for systems that are designed to operate autonomously, without frequent user input once they have been turned on. We have already found several examples of this in the programs that we have been writing to control Timbot, both in small components, and in complete programs, which we can express in timber by using the following `Controller` abstraction:

```
record Controller =
  start :: Action
  stop  :: Action
```

4.1 An Acceleration Controller: First Attempt

As an example, the following code defines a controller that will set a `timbot` in motion, accelerating from rest by increasing the vehicle's speed by `incr` units after each period of time `t`, but never exceeding the specified `maxSpeed`.

```
accelControl :: Speed -> Speed -> TimeDiff -> Timbot -> Template Controller
accelControl maxSpeed incr t timbot
 = template running := False
   in let accel s = action if running then
                             timbot.speed s
                             let s' = s + incr
                             if s' <= maxSpeed then
                               after t (accel s')
   in record start = action if not running then
                             running := True
                             accel 0
             stop  = action running := False
                            timbot.speed 0
```

This controller is useful in practice because it reduces the possibility of a damaging jolt that could occur if we set the speed of `Timbot` directly to the target speed. The key here is the `accel` method that is called with a zero speed setting when the controller is first started. Subsequent recursive calls increase the

speed in steps using `after` to ensure that they are distributed correctly over time. Additional logic, using a Boolean state variable `running`, will terminate the acceleration if the controller is stopped before the target speed is reached.

Unfortunately, our definition has a serious flaw: if the controller is turned off, but then turned back on again before the next `accel` step is executed, then we will continue with the sequence of `accel` calls initiated when the controller was first started, while also generating a second sequence of calls for the later start. Such behavior is unlikely to produce satisfactory results!

4.2 An Acceleration Controller: Second Attempt

Clearly, it is not enough for an acceleration controller's `stop` method just to reset the running flag and bring the vehicle to a stop with `timbot.speed 0`; it must also cancel any pending calls to `accel`. It is easy to implement this using standard Timber library functions. But we will go a step further and generalize to obtain an abstraction that can be used in other contexts, while also neatly encapsulating our solution to the bug in the original `accelControl`. The benefit, of course, is that other programmers can then use this more general construct to build new controllers more concisely, without recreating our original bug.

We start with the definition of a new subtype of `Controller` that adds a method called `invoke`. This new method will be invoked immediately after the controller is started. Each time it is called, however, it returns a value of type `Maybe TimeDiff`, indicating when (if at all) it should next be invoked.

```
record RepeatController < Controller =
  invoke :: Request (Maybe TimeDiff)
```

Now we can use an object of this type to build a controller with the correct behavior using the following `startstop` function:

```
startstop :: RepeatController -> Template Controller
startstop rc
 = template running := False
            sc <- singlecall
   in let tick = action mpa <- sc.invoke
                        case mpa of
                           Just t  -> sc.call (after t tick)
                           Nothing -> done
   in record start = action if not running then
                               running := True
                               rc.start
                               tick
              stop  = action if running then
                               running := False
                               sc.cancel
                               rc.stop
```

Note the use of the single call object `sc`, which allows our program to schedule the execution of an action `a` using `sc.call a`, but also allows that action to

be canceled, if it has not already started, using `sc.cancel`. This provides the missing feature that we needed to avoid the original `accelControl` bug, and is included as part of the Timber libraries.

Now, for example, we can recode our `accelControl` controller more concisely, and without the bug, using the following definition:

```
accelControl maxSpeed incr t timbot
 = template s    := 0
            ctrl <- startstop (record
                      start  = action s := 0
                      stop   = action timbot.speed 0
                      invoke = request
                        timbot.speed s
                        s := s + incr
                        return (if s <= maxSpeed
                                  then Just t else Nothing))
   in ctrl
```

Notice that all of the logic associated with the `running` flag has been captured and hidden away by the use of `startstop`.

4.3 Imperative versus Declarative: A Matter of Style?

Our definitions of `accelControl` have an imperative feel, which some readers may feel detracts from the goals of Timber as a declarative language. This is subjective, but we note also that it often comes down to a debate about programming style. The following alternative definition, for example, while retaining some imperative elements, avoids the explicit recursion in the original:

```
accelControl maxSpeed incr t timbot
 = let profile = zip [0,t..] [0,incr..maxSpeed]
   in template mc <- multicall
   in record start = action
                       forall (ti,si) <- profile do
                         mc.call (after ti (timbot.speed si))
              stop  = action
                       mc.cancel
                       timbot.speed 0
```

This code defines a list of (time,speed) pairs called `profile` that describes the complete acceleration process. This list is used when the controller is started to generate a corresponding sequence of (time-delayed) actions. (The `forall` construct—which might suggest a loop in an imperative language—is really just convenient syntactic sugar for a standard operation on lists.) The only real difference here is the use of a `multicall`, which behaves much like a `singlecall`, except that it enables us to call (and subsequently cancel) a collection of multiple pending actions. The `multicall` method used here is not included in the current Timber libraries, but will perhaps be added in a future version.

4.4 Other Controller Components

As we write programs to control Timbot, we are collecting a library of reusable components, like `accelControl`, that are useful in other applications. In this section, we describe three representative examples from this growing collection.

Our first example is a `periodicControl`, which will execute a particular command at regular intervals for as long as the controller is turned on. Its definition is a simple application of `startstop`:

```
periodicControl      :: TimeDiff -> Cmd a -> Template Controller
periodicControl t cmd = do rc <- template in
                                 record start  = action done
                                        stop   = action done
                                        invoke = request cmd
                                                         return (Just t)
                           startstop rc
```

No special actions are needed (beyond those already handled by `startstop`) when a `periodicControl` component is either started or stopped, so the trivial action, `done`, is specified for these two methods.

Our second example is `sweepControl`, which can be used to sweep a device (such as the sonar, or the camera) across a range of different angles (between `minA` and `maxA`), changing the angle by some fixed `incr` after each `t` units of time, and triggering an appropriate action at each point. For this example, we use an object with a state variable `angle` that records the current angle, and a Boolean state variable `incr` to indicate if the angle is currently increasing or not (i.e., moving from `minA` to `maxA` or from `maxA` to `minA`). As one might hope, the periodic stepping of `sweepControl` is captured naturally using `periodicControl`.

```
sweepControl (minA, maxA, stepA, t) act
 = template
     angle := minA
     incr  := True
     ctrl  <- periodicControl t
               (action act angle
                       if incr then angle := angle + stepA
                                    if angle > maxA then
                                      angle := maxA
                                      incr  := False
                               else angle := angle - stepA
                                    if angle < minA then
                                       angle := minA
                                       incr  := True)
    in ctrl
```

Almost all of the code here deals with the specific needs of a `sweepControl` component—much of which has to do with calculating how the sweep angle should change from one step to the next. There is, by comparison, very little in the way of boilerplate code, because that has already been packaged away for us in abstractions like `periodicControl`.

Our third example demonstrates a different style of definition. In this case, a `multiControl` controller can be used to start/stop each of the elements in a list of controllers from a single start/stop command.

```
multiControl   :: [Controller] -> Template Controller
multiControl cs = template in record
                    start = action forall c <- cs do
                                    c.start
                    stop  = action forall c <- cs do
                                    c.stop
```

The following simple program illustrates how the components described above can be combined to construct simple control programs for Timbot. This particular section of code, for example, constructs independent sweep controllers, each operating at a different frequency, for the camera pan and tilt, and then uses `multiControl` to package them into a single controller.

```
do cam <- getTimbot env
   pc  <- sweepControl (-60, 60, 6, 50*milliseconds) cam.pan
   tc  <- sweepControl (-30, 30, 2, 80*milliseconds) cam.tilt
   multiControl [pc, tc]
```

In this example, we are simply using Timber to describe, at a high-level, the construction and connections between a group of reusable control components. What the language hides are the subtle and sometimes complex concurrency, scheduling, and synchronization issues that are needed to weave the code from each component together with the intended timing.

5 Control Applications

In this section we describe some simple control programs that can be constructed from the components in previous sections. In particular, these programs are designed to use information obtained from sonar. The most important details to notice in these examples are the ease with which components can be combined, and the clarity that results from the implicit treatment of concurrency. Note also that each example is packaged using the same controller abstraction as the components from which they are built. As a result, these examples could in turn be reused as components in a larger, more complex system.

5.1 Simple Obstacle Avoidance

In this section we show the code for a simple obstacle avoidance program that drives the robot forward while sweeping the sonar across the path in front of it to look for obstacles. (For the purposes of this paper, an obstacle is any object that is within 1m of the robot on its forward path. In practice, we often test Timbot by stepping into the path of the robot and using ourselves as obstacles!) If an obstacle is detected, then the robot will stop, but continue scanning in the

hope that the obstacle will move. If a full second passes with no obstacle being detected, then the robot will once again begin accelerating forward again.

The code for `simpleObstacleAvoid` is straightforward, obtaining a `timbot` interface; building an accelerator controller, a listener for the sonar, and a controller to sweep and trigger the sonar; and gluing these pieces together.

```
simpleObstacleAvoid env
 = do timbot <- getTimbot env
      accel  <- accelControl 50 10 (500*milliseconds) timbot
      lstn   <- obstacleLstn accel
      sweep  <- sweepControl (-12, 12, 2, 100*milliseconds)
                  (\a -> timbot.angle a lstn)
      return sweep
```

The main control logic is provided by the listener that receives distance measurements from the sonar, which we construct using the `obstacleLstn` function:

```
obstacleLstn :: Controller -> Template SonarListener
obstacleLstn ctrl
  = template lastTime := 0
    in record distance d
                = action t <- currentBaseline
                          case d of
                            Just x | x<1.0 -> lastTime :=t
                                              ctrl.stop
                            _              -> if t > lastTime+1*seconds
                                                then ctrl.start
```

This listener records the time at which an obstacle was last detected in a private state variable `lastTime`. Each time the sonar reports a distance, the listener checks to see if it indicates the presence of an obstacle. Note that an out-of-range reading is treated, perhaps rather dangerously, as an indication that no obstacle has been seen! The careful reader might also spot that `obstacleLstn` is a candidate for reuse because it can be connected to an arbitrary controller, and not just to the `accelControl` that is used in `simpleObstacleAvoid`.

5.2 Wall Following

In this section, we show the Timber code for another well-known example of autonomous robot control: wall-following. The goal of this application is to drive the robot along at a fixed distance from a wall on its right hand side. If the robot gets too close to the wall (below a distance `minD`), then we steer the robot to the left, and away from the wall. On the other hand, if it gets too close (greater than a distance `maxD`), then it will steer to the right, and toward the wall. (For readers not familiar with this particular problem, we should note that this simple strategy will only work correctly within certain parameters—we assume that the vehicle begins parallel to the wall at a distance within `minD` and `maxD`, and that it does not move or turn too quickly.)

Again, the code breaks into two pieces, the first of which constructs and connects components, while the second encodes the main logic in a listener.

```
wallFollow minD maxD env
 = do timbot  <- getTimbot env
      accel   <- accelControl 40 10 (500*milliseconds) timbot
      lstn    <- wallLstn timbot accel minD maxD
      trigger <- periodicControl (200*milliseconds)
                                 (timbot.angle 70 lstn)
      multiControl [trigger, accel]

wallLstn :: Timbot -> Controller -> Distance -> Distance
                -> Template SonarListener
wallLstn timbot ctrl minD maxD
 = template in record
     distance d = action
       case d of Just x  | x < minD  -> timbot.steer hardLeft
                         | x > maxD  -> timbot.steer hardRight
                         | otherwise -> timbot.steer 0
                 Nothing             -> ctrl.stop
```

The listener that we have used in this example responds a little differently, and perhaps too cautiously, to an out-of-range reading from the sonar (the case for `Nothing` in the definition above) by assuming that this indicates the end of the wall, and so bringing the vehicle to rest. In fact, an out-of-range reading might also have been the result of a transient glitch. We leave it as an exercise to the reader to extend the definition here to delay stopping the vehicle until several consecutive out-of-range readings have been received, and so reduce the chance that the robot might stop prematurely,

6 Future and Related Work

The examples in this paper have demonstrated how Timber can be used to support an elegant and compositional approach to the construction of simple control algorithms for the Timbot robot vehicle. The high-level treatment of concurrency is particularly useful in avoiding the need for programmers to deal explicitly with the thorny issues of synchronization, scheduling, etc. As we continue to develop more interesting and more sophisticated control programs, we are also building a useful library of flexible and reusable control components.

There have been several other attempts to explore the use of declarative languages in similar application domains. Rees and Donald [9], for example, showed how the abstraction mechanisms of Scheme can be used in robot control, but also relied on explicit concurrency and synchronization. Wallace and Runciman [10] showed how functional languages can be used to describe an embedded controller for a lift shaft, but adopted a more primitive process model that allows processes to receive only one type of message. Most recently, *Functional Reactive Programming* (FRP) has been used to provide declarative specifications of

event-based programs with continuously time-varying *behavior* functions. The FRP style has been used in a number of applications including robot control [6, 5], where a special *task* monad is used to sequence tasks and track the robot state. More detailed comparison of Timber and FRP is a topic for future work.

Acknowledgments

The work reported in this paper was sponsored in part by DARPA, contract #F33615-00-C-3042, as part of the PCES program (Program Composition for Embedded Systems). This work has benefited from the comments of members of the Project Timber team, and of the PacSoft and SySL centers at OGI. Particular thanks: to Perry Wagle for considerable assistance in building Timbot, and for suggesting and prototyping interesting control applications; and to Andrew Black, Dick Kieburtz, and James Hook for helpful insights and encouragement.

References

[1] Andrew P. Black, Magnus Carlsson, Mark P. Jones, Richard Kieburtz, and Johan Nordlander. Timber: A programming language for real-time embedded systems. http://www.cse.ogi.edu/PacSoft/projects/Timber/, April 2002.

[2] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML - Revised.* MIT Press, 1997.

[3] Johan Nordlander. *Reactive Objects and Functional Programming.* PhD thesis, Department of Computer Science, Chalmers University of Technology, Göteborg, Sweden, May 1999.

[4] Johan Nordlander, Mark Jones, Magnus Carlsson, Dick Kieburtz, and Andrew Black. Reactive objects. In *Proceedings of the Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Arlington, VA, April 2002.

[5] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *Proceedings of the IEEE International Conference on Robotics and Automation*, Detroit, MI, May 1999.

[6] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Proceedings of Principles and Applications of Declarative Languages (PADL '99)*. Springer-Verlag, 1999.

[7] Simon Peyton Jones and John Hughes, editors. *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*, 1999. Available from `http://www.haskell.org/definition/`.

[8] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th Symposium on Principles of Programming Languages (POPL '93)*. ACM, January 1993.

[9] Jonathan A. Rees and Bruce R. Donald. Program mobile robots in scheme. In *Proceedings of ICRA '92, the IEEE International Conference on Robotics and Automation*, 1992.

[10] Malcolm Wallace and Colin Runciman. Lambdas in the liftshaft - functional programming and an embedded architecture. In *Proceedings of Functional Programming and Computer Architecture, (FPCA '95)*, La Jolla, California, June 1995. ACM Press.

APPENDIX G

Priority-Progress Streaming for Quality-Adaptive Multimedia. Buck Krasic and Jonathan Walpole, *In Proceedings of the ACM Multimedia Doctoral Symposium, Ottawa, Canada, October 2001.*

Priority-Progress Streaming for Quality-Adaptive Multimedia*

Charles Krasic†
Oregon Graduate Institute
20000 NW Walker Rd.
Beaverton, Oregon 97206
krasic@cse.ogi.edu

Jonathan Walpole‡
Oregon Graduate Institute
20000 NW Walker Rd.
Beaverton, Oregon 97206
walpole@cse.ogi.edu

ABSTRACT

The Internet's ubiquity amply motivates us to harness it for video distribution, however, its best-effort service model is in direct conflict with video's inherent timeliness requirements. Today, the Internet is unrivaled in its rich composition, consisting of an unparalleled assortment of networks and hosts. This richness is the result of an architecture that emphasizes interoperability over predictable performance. From the lowest levels, the Internet architecture prefers the best effort service model. We feel current solutions for media-streaming have yet to adequately address this conflict between timeliness and best-effort service.

We propose that streaming-media solutions targetted at the Internet must fully embrace the notion of graceful degradation, they must be architected with the expectation that they operate within a continuum of service levels, adjusting quality-resource trade-offs as necessary to achieve timeliness requirements. In the context of the Internet, the continuum of service levels spans across a number of time scales, ranging from sub-second timescales to timescales as long as months and years. We say sub-second timescales in relation to short-term dynamics such as network traffic and host workloads, while timescales of months and years relate to the continuous deployment of improving network, compute and storage infrastructure.

We support our thesis with a proposal for a streaming model which we claim is simple enough to use end-to-end, yet expressive enough to tame the conflict between real-time and best-effort personalities of Internet streaming. The model is called Priority-Progress streaming. In this proposal, we will describe the main features of Priority-Progress streaming, which we have been implemented in a software-based streaming video system, called the Quasar pipeline.

Our work is primarily concerned with the class of streaming applications. To prevent confusion, we now clarify the important distinction between streaming and other forms of distribution, namely download. For a video, we assume download is defined so that the transfer of the video must complete before the video is viewed. Transfer and viewing are temporally sequential. With this definition, it is a simple matter to employ Quality-adaptive video. One algorithm would be to deliver the entire video in the order from low to high quality components. The user may terminate the download early, and the incomplete video will automatically have as high quality as was possible. Thus, Quality-adaptive download can be implemented in an entirely best-effort, time-insensitive, fashion. On the other hand, we assume streaming means that the user views the video at the same time that the transfer occurs. Transfer and viewing are concurrent. There are timeliness requirements inherent in this definition, which can only be reconciled with best-effort delivery through a time-sensitive adaptive approach.

1. PRIORITY-PROGRESS STREAMING

The central notion of Priority-Progress streaming is to decompose application data into units of work, application data units (ADUs), each labeled with timestamp and priority. The timestamp is meant to capture the timeliness requirements of each ADU, and is expressed in units of the *normal play time* of the media stream. As ADUs are processed end-to-end, the timestamps and priorities provide vital information necessary to regulate work so as to ensure proper real-time progress. The priority exposes the layered nature of the media, where quality can be progressively improved given more of the limiting resource: network, processing, or storage.

We use the priorities to achieve graceful degradation. Although a threshold priority-drop approach could be applied rather directly to match quality to available resources, there remains an issue that the resource-quality relationship may vary rapidly, and a user will likely be annoyed by the unstable quality. Our own experience implementing a Quality-Adaptive video system has shown us that streams can have non-smooth and highly dynamic quality-rate relationships, which are inconsistent across resource types[3]. Furthermore, the experience of others indicates to us that predict-

*This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

†Phd student

‡Phd advisor

ing available network bandwidth is equally problematic[4]. We are thus motivated to reformulate the problem to avoid explicitly predicting either priority-rate relationship or resource availability. The unique aspect of Priority-Progress streaming presented here is that it uses ADU-reordering in its buffers to do just that.

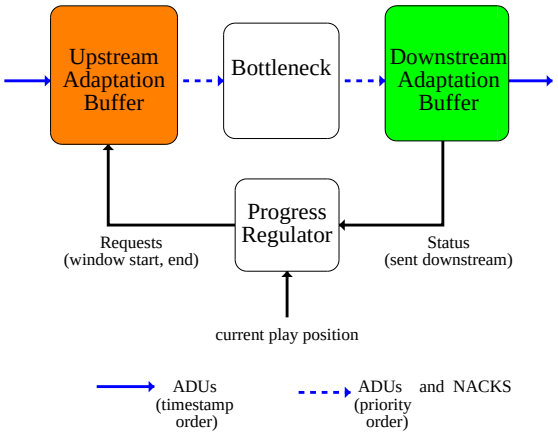


Figure 1: Priority-Progress Control

Figure 1 depicts the structure of Priority-Progress control within a media pipeline. A pair of re-ordering buffers is employed around each *bottleneck* pipeline component. For example, in the Quasar pipeline we have one such element responsible for streaming across a network transport. Similarly, the software-decompression element is considered a bottleneck, as it has unpredictable progress rates due both to data dependencies in MPEG and to external influences from competing tasks in a multi-tasking environment. The capacities of the re-ordering buffers are managed in terms of time, making use of the timestamp labels on ADUs[1]. The algorithm for Priority-Progress Streaming contains three subcomponents, for the upstream buffer, downstream buffer, and progress regulator respectively.

The upstream and downstream buffer algorithms operate as follows. The upstream buffer admits all ADUs within the time boundaries provided by the progress regulator, these boundaries delimit the *adaptation window*. Each time the regulator advances the window forward, the unsent ADUs from the old window position are expired and the window is populated with ADUs of the new position. ADUs flow from the buffer in priority-order through the bottleneck to the downstream adaptation buffer, as fast as the bottleneck will allow. The downstream adaptation buffer collects ADUs and re-orders them to timestamp order. ADUs are allowed to flow out from the downstream buffer when it is known that no more ADUs for a timestamp are coming.

To explain how the progress regulator works, it is important to understand how the flow of ADUS relates to the presentation timeline, as shown in Figure 2. The timeline is based on the usual notion of *normal play time*, where a presentation is thought to start at time zero (epoch a) and run to its duration (epoch e). Once started, the presentation

[1]For simplicity, we consider the buffers unbounded in terms of space, although space constraints can be enforced without difficulty.

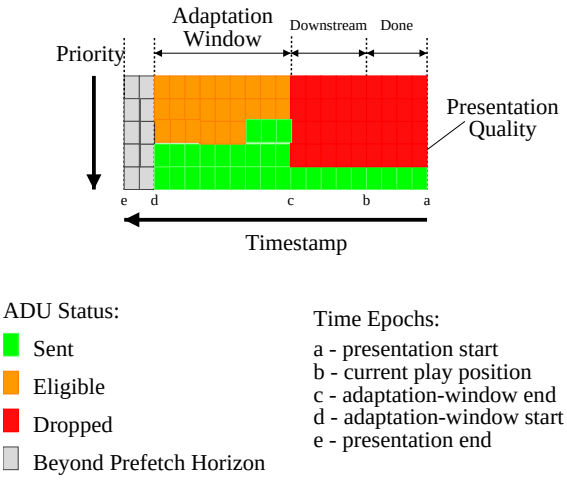


Figure 2: Priority-Progress Timeline

time (epoch b) advances at some rate synchronous with real-time. The ADUs within the adaptation window in the timeline correspond to the contents of the upstream and downstream re-order buffers; ADUs within the adaptation window that are *sent* are either in the bottleneck or the downstream buffer, while ADUs still *eligible* are in the upstream buffer. The interval spanned by the adaptation window is the key to our ability to control the responsiveness-stability trade-off of quality adaptation. The larger the interval, the less responsive and the more stable quality adaptation will be. A highly responsive system is generally required at times of interactive events (start, fast-forward, etc.), while stable quality is generally preferable. We transition from responsiveness to stability by progressively expanding the adaptation window. The regulator can manipulate the size of the window through actuation of the ratio between the rate at which the adaptation window is advanced and the rate at which the presentation clock advances. By advancing the timeline faster than the presentation clock (ratio > 1), the regulator can expand the window with each advancement, skimming some current quailty in exchange for more stable quality later.

2. REFERENCES

[1] W. chi Feng, M. Liu, B. Krishnaswami, and A. Prabhudev. A priority-based technique for the best-effort delivery of stored video. In *SPIE/IS&T Multimedia Computing and Networking 1999*, San Jose, California, January 1999.

[2] N. Feamster, D. Bansal, and H. Balakrishnan. On the interactions between layered quality adaptation and congestion control for streaming video. In *11th International Packet Video Workshop (PV2001)*, Kyongiu, Korea, April 2001.

[3] C. Krasic and J. Walpole. QoS scalability for streamed media delivery. CSE Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.

[4] R. Rejaie, M. Handley, and D. Estrin. Quality adaptation for congestion controlled video playback over the internet. In *Proceedings of ACM SIGCOMM '99 Conference*, Cambridge, MA, October 1999.

APPENDIX H

Supporting Low-Latency TCP-Based Media Streams. Ashvin Goel, Buck Krasic, Kang Li, Jonathan Walpole, In Proceedings of the Tenth International Workshop on Quality of Service (IWQoS 2002), Miami Beach, Florida, May 2002.

Supporting Low Latency TCP-Based Media Streams

Ashvin Goel Charles Krasic Kang Li Jonathan Walpole
Oregon Graduate Institute, Portland
{ashvin,krasic,kangli,walpole}@cse.ogi.edu

Abstract— **The dominance of the TCP protocol on the Internet and its success in maintaining Internet stability has led to several TCP-based stored media-streaming approaches. The success of these approaches raises the question whether TCP can be used for low-latency streaming. Low latency streaming allows responsive control operations for media streaming and can make interactive applications feasible. We examined adapting the TCP send buffer size based on TCP's congestion window to reduce application perceived network latency. Our results show that this simple idea significantly improves the number of packets that can be delivered within 200 ms and 500 ms thresholds.**

I. Introduction

Traditionally, the multimedia community has considered TCP unsuitable for streaming audio and video data. The main issues raised against TCP-based streaming have been related to *congestion control* and *packet retransmissions*. TCP congestion control is designed to probe available bandwidth through deliberate manipulation of the transmission rate. This rate variation can impede effective streaming because the streaming requirements are not necessarily matched with the transmission rate, causing either data dropping or accumulation of buffered data and thus delay. In addition, congestion control can lead to sustained or long-term reduction in rate.

TCP uses packet retransmissions to provide in-order, lossless packet delivery. Packet retransmissions can potentially introduce unacceptable end-to-end latency and thus re-sending media data may not be appropriate because it would arrive too late for display at the receiver.

Recently, several approaches have been proposed to overcome these problems [4], [26], [14], [25], [18]. These TCP-based stored media streaming approaches use a combination of *client-side buffering* and efficient *QoS adaptation* of the streamed data. Client-side buffering essentially borrows some current bandwidth to prefetch data to protect against future rate reduction. Thus, with sufficient client-side buffering, short-term rate variations introduced by TCP as well as the delay introduced by packet retransmissions can both be handled. QoS adaptation allows fine-grained adjustment of the rate-distortion tradeoff, i.e., rate versus quality adjustment, during the transmission process and thus allows handling long-term rate changes by adjusting quality dynamically.

TCP-based streaming is desirable because TCP offers several well known advantages. TCP provides congestion controlled delivery which is largely responsible for the remarkable stability of the Internet despite an explosive growth in traffic, topology and applications [13]. TCP handles flow control and packet losses, so applications do not have to worry about recovery from packet losses. This issue is especially important because the effects of packet loss are non-uniform and can quickly become severe. For instance, loss of the header bits of a picture typically renders the whole picture and possibly a large segment of surrounding video data unviewable while loss of certain pixel blocks may be virtually imperceptible. Thus media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [27] that potentially have high bandwidth and processing overhead. Finally, given the large TCP user base, there is great interest in improving its performance. Such improvements can also help media streaming.

In this paper, we study the feasibility of using TCP for low-latency media streaming. We are concerned with *protocol latency*, which we define as the time difference from a write on the sender side to a read on the receiver side, i.e., socket to socket latency. Low latency streaming is desirable for several applications. For streaming media, control operations such as the sequence of start play, fast forward and restart play become more responsive because the network and the end-points have low delay in the data path. For video on demand servers, low latency streaming offers faster channel surfing (starting and stopping of different channels). Similarly, multimedia document browsing becomes more responsive. Finally, with sufficiently low latency streaming, interactive streaming applications become feasible.

Although there have been several studies that describe the packet delays experienced by TCP flows [1], [23], [9] there has been much less work describing the protocol latency observed by applications streaming over TCP. This lack of study of protocol latency is partly because TCP has often been considered impractical for streaming applications and thus few TCP-based streaming applications have been developed. In addition, non-QoS adaptive streaming applications require large buffering at the ends to handle bandwidth variations, so protocol latency can be a second order effect. Fortunately, with quality adaptive streaming applications, the buffering needed at the end-points can be tuned and made small and thus protocol latency becomes more significant.

This paper examines TCP protocol latency by showing the latency observed at the sender side, receiver side and the network under various network conditions. Our results show that, surprisingly, a significant portion of the protocol latency occurs due to TCP's *send buffer* and this latency can be eliminated by making some simple send-buffer modifications to the sender side TCP stack without changing the TCP protocol in any way. These modifications dynamically adapt (reduce) the send buffer size and have similarity to the send-buffer tuning work by Semke [29]. However, unlike their work which focuses on improving TCP throughput, this work focuses on reducing

This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

socket to socket latency.

Our experiments show that these modifications reduce the average protocol latency to well within the interactive latency limits of approximately 200 ms [12] when the underlying network round-trip time is less than 100 ms (coast-to-coast round-trip time in the US [9]).[1] This reduction in latency comes at a small expense in throughput.

At this point, it may seem that our send-buffer reduction approach would reduce latency from the TCP layer but would re-introduce it at the application layer, and thus the net effect on application-level end-to-end latency is unclear. Fortunately, this issue is not a real problem because we assume that latency-sensitive applications are 1) quality-adaptive and 2) they use `poll` and non-blocking `write` calls on the sending side. The benefit of low latency streaming is that the sending side can wait longer before making its quality adaptation decisions, i.e., it has more control and flexibility over what data should be sent and when it should be sent. For instance, if the low protocol-latency network doesn't allow the application to send data for a long time, the sending side can drop low-priority data and then send data, which will arrive at the receiver with low delay (instead of committing the low-priority data to a large TCP send-buffer early and then lose control over quality adaptation when that data is delayed in the send buffer). The non-blocking write calls ensure that the sending side is not blocked from doing other work (such as media encoding) while the network is busy. In addition, the application does not spend CPU cycles polling for the socket-write ready condition since the kernel informs the application when the socket is ready for writing.

The sender-side modifications reduce average protocol latency significantly but are not sufficient for interactive streaming applications since many packets can still observe latencies much higher than 200 ms. These latency spikes occur due to packet dropping and retransmissions and thus motivate the need for mechanisms that reduce packet dropping in the network. One such mechanism is explicit congestion notification (ECN) [24]. With ECN, routers use active queue management [5] and indirectly inform TCP of impending congestion by setting an ECN bit on packets that would otherwise have been dropped. TCP uses the ECN bit to pro-actively reduce its sending rate, thus reducing network load and packet dropping in the network. This paper explores how TCP enabled with ECN effects protocol latency.

The next section presents our modifications to the TCP sending side to reduce protocol latency. Section III describes our experimental methodology for evaluating the latency behavior of TCP. Section IV presents our results. Section V summarizes related work in multimedia and low latency streaming, and TCP congestion control. Section VI discusses future work in low-latency TCP streaming, and finally, Section VII presents our conclusions.

II. TCP Send Buffer

This section discusses our approach to reducing protocol latency by dynamically adjusting the TCP *send buffer* size. TCP is a window-based protocol, where its window size is the maximum number of distinct (and unacknowledged) packets in flight in the network at any time. TCP adapts the size of its window based on congestion feedback and stores this size value in the TCP variable CWND. TCP uses a *fixed size* send buffer to store application data before the data is transmitted. This buffer has two functions. First, it handles rate mismatches between the application sending rate and TCP's transmission rate. Second, it is used to keep copies of the packets in flight (its current window) so they can be retransmitted when needed. Since CWND stores the number of packets in flight, its value can never exceed the send buffer size.

From a latency perspective, the fixed size send buffer can introduce significant latency into the TCP stream. As a concrete example, the send buffer in most current Unix kernels is at least 64KB. For a 300 Kbs video stream, a full send buffer contributes 1700 ms of delay. By comparison, the round trip delay may lie between 50-100 ms for coast-to-coast transmission within the United States. In addition, the buffering delay increases for smaller bandwidth streams or with increasing competition since the stream bandwidth goes down.

We believe that for latency sensitive streams, sender-side buffering should be moved out of the TCP stack and applications should be allowed to handle buffering as much as possible. This approach is in keeping with the end-to-end principle followed by TCP where the protocol processing complexity is moved out of the network as much as possible to the stream end points. We do not modify TCP receive-side buffering because our applications aggressively remove data from the receive-side buffer. Thus, receive-side delay is only as issue when packets are retransmitted by TCP. This issue is discussed further in Section IV-C.

A. Adapting Send Buffer Size

One method for reducing the latency caused by the send buffer is to statically reduce the size of the send buffer. This approach has a negative effect on the throughput of the flow if the number of packets in flight (CWND) is limited by the send buffer (and not by the network congestion signal). In this case, the flow throughput is directly proportional to the send buffer size and decreases with a smaller send buffer. We reject this approach because although our main goal is to reduce protocol latency, we also aim to achieve throughput comparable to standard TCP.

Now suppose that the send buffer was sufficiently large that TCP could adjust the value of CWND based only on congestion (and receiver buffer) feedback. It should be clear that for this condition to hold, the size of the send buffer should be at least CWND packets. A smaller value would limit CWND to the send buffer size and reduce the throughput of the flow. A larger value should not affect throughput significantly since TCP would not send more than CWND packets anyway. However, a larger value increases protocol latency because only CWND packets can be in flight at any time, and thus the rest of the packets have to sit in the send buffer until acknowledgments have been received for the previous packets.

This discussion shows that adjusting the send buffer size to follow CWND can reduce protocol latency without signif-

[1] We are focusing on protocol latency (or socket to socket latency) and ignore the processing times at the application end points in this paper.

icantly affecting flow throughput. We have implemented this approach, as described in Section II-B. This approach impacts throughput when TCP could have sent a packet but there are no new packets in the send buffer. This condition can occur for several reasons. First, with each acknowledgment arrival, standard TCP has a packet in the send buffer that it can send immediately. If the send buffer size is limited to CWND, then TCP must inform the application and the application must write the next packet before TCP can send it. Thus, system timing and scheduling behavior can affect TCP throughput. Second, back-to-back acknowledgment arrivals exacerbate this problem. Finally, the same problem occurs when TCP increases CWND. These adverse affects on throughput can be reduced by adjusting the buffer size so that it is larger than CWND. To study the impact on throughput, we experimented with three different send buffer configurations as described in the next section.

B. Send Buffer Modifications

To reduce sender-side buffering, we have made a small send-buffer modification to the TCP stack on the sender side in the Linux 2.4 kernel. This modification can be enabled per socket by using a new SO_TCP_MIN_BUF option, which limits the send buffer size to $A*$CWND+MIN$(B,$ CWND$)$ packets at any given time. The send buffer size is at least CWND because A must be an integer greater than zero and B is zero or larger. We assume, as explained in more detail later, that the size of each application packet is MSS (maximum segment size). With the send-buffer modification, an application is blocked from sending when there are $A *$ CWND $+$ MIN$(B,$ CWND$)$ packets in the send buffer. In addition, the application is woken up when at least one packet can be admitted in the send buffer. By default A is one and B is zero, but these values can be made larger with the SO_TCP_MIN_BUF option. From now on, we call a TCP stream that has the SO_TCP_MIN_BUF option turned on with parameters A and B, a MIN_BUF(A, B) stream.

With these modifications to TCP and assuming a MIN_BUF$(1, 0)$ stream, the send buffer will have at most CWND packets after an application writes a packet to the socket. TCP can immediately transmit this packet since this packet lies within TCP's window. After this transmission, TCP will again allow the application to write data. Thus as long as CWND is non-decreasing, TCP will not add any buffering delay to a stream. Delay is added only during congestion when TCP decreases the value of CWND. Our experiments in Section IV show that this delay is generally much smaller than the standard TCP send-buffer delay.

The SO_TCP_MIN_BUF option exposes the parameter A and B, because they represents a tradeoff between latency and throughput. Larger values of A or B add latency but can improve throughput as explained in the previous section. We experimented with three MIN_BUF streams: MIN_BUF$(1, 0)$, MIN_BUF$(1, 3)$ and MIN_BUF$(2, 0)$. These streams should have increasing latency and throughput. A MIN_BUF$(1, 0)$ stream is the default stream with the least protocol latency. We expect a MIN_BUF$(2, 0)$ stream to have the same throughput as TCP because there are CWND extra packets in the send buffer and even if acknowledgments for all packets in the previous window come simultaneously, the next window of packets can be sent without first getting packets from the application. Thus a MIN_BUF$(2, 0)$ stream should behave similarly (in terms of throughput) to a TCP stream [16]. Finally, we chose a MIN_BUF$(1, 3)$ stream to see how three extra packets affect latency and throughput. If no more than three acknowledgments arrive back to back, then this stream should behave similar to TCP in terms of bandwidth. Section IV presents latency and throughput results for the three streams. Briefly, our results show that 1) MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows has similar latencies and these latencies are much smaller than MIN_BUF$(2, 0)$ or TCP flows, and 2) while a MIN_BUF$(1, 0)$ flow suffers 30 percent bandwidth loss, the MIN_BUF$(1, 3)$ flow suffers less than 10 percent bandwidth loss. Thus, the MIN_BUF$(1, 3)$ flow represents a good latency-bandwidth compromise.

1) Sack Correction: The previous discussion about the send buffer limit applies for a non-SACK TCP implementation. For TCP SACK [15], we make a *sack correction* by adding an additional term *sacked_out* to $A *$ CWND $+$ MIN$(B,$ CWND$)$. The sacked_out term (or an equivalent term) is maintained by a TCP SACK sender and is the number of selectively acknowledged packets. With TCP SACK, when selective acknowledgments arrive, the packets in flight are no longer contiguous but lie within a CWND+sacked_out packet window. We make the sack correction to ensure that the send buffer limit includes this window and is thus at least CWND+sacked_out. Without this correction, TCP SACK is unable to send new packets for a MIN_BUF flow and assumes that the flow is application limited. It can thus reduce the congestion window multiple times after the arrival of selective acknowledgments.

2) Alternate Application-Level Implementation: It is conceivable that the objectives of the send-buffer modifications can be achieved at the application level. Essentially the application would stop writing data when the socket buffer has a fill level of $A *$ CWND $+$ MIN$(B,$ CWND$)$ packets or more. The problem with this approach is that the application has to poll the socket fill level. Polling is potentially both expensive in terms of CPU consumption and inaccurate since the application is not informed immediately when the socket-fill level goes below the threshold.

C. Application Model

In this paper, we are concerned with protocol latency. We ignore the processing time at the application end points since these times are application dependent. However, these times must also be included when studying the feasibility of a low latency application such as an interactive media streaming application.

We assume that latency-sensitive applications use non-blocking read and write socket calls. The protocol latency is measured from when the packet write is initiated on the sender side to when the same packet is completely read on the receiver side. The use of non-blocking calls generally means that the application is written using an event-driven architecture [22].

We also assume that applications explicitly align their data with packets transmitted on the wire (application level framing) [2]. This alignment has two benefits: 1) it minimizes any latency due to coalescing or fragmenting of packets below

the application layer, 2) it ensures that low-latency applications are aware of the latency cost and throughput overhead of coalescing or fragmenting application data into network packets. For alignment, an application writes MSS (maximum segment size) sized packets on each write. TCP determines MSS during stream startup but the MSS value can change due to various network conditions such as routing changes [17]. A latency-sensitive application should be informed when TCP determines that the MSS has changed. Currently, we detect MSS changes at the application level by querying TCP for the MSS before each write. Another more efficient option would be to return a write error on an MSS change for a MIN_BUF socket.

III. Experiments

In this section, we describe the tests we performed to evaluate the latency and throughput behavior of standard TCP and MIN_BUF streams under various network conditions. All streams use TCP SACK and MIN_BUF streams use the sack correction described in Section II-B. We performed our experiments on a Linux 2.4 test-bed that simulates WAN conditions by introducing delay at an intermediate Linux router in the test-bed.

A. Experimental Scenarios

The first set of tests considers the latency response of TCP streams to a sudden increase in congestion. Increase in congestion is triggered with three types of flows: 1) competing long-lived TCP flows, 2) a flash crowd of many small TCP flows, and 3) a competing constant bit rate (CBR) flow, such as a UDP flow. The long-lived competing flows are designed to simulate other streaming traffic. The flash crowd of short TCP flows simulates web transfers. In our experiments, the small flows have fixed packet sizes and they are run back to back so that the number of active TCP connections is roughly constant [11]. The CBR flow simulates non-responsive UDP flows.

While these traffic scenarios do not necessarily accurately model reality, they are intended to explore and benchmark the latency behavior of TCP and MIN_BUF streams in a well characterized environment. These tests are designed to emulate a heavily loaded network environment.

The second set of tests measures the relative throughput share of TCP and MIN_BUF streams. Here we are mainly concerned with the bandwidth lost by MIN_BUF traffic. These experiments are performed with the same types of competing flows described above.

We are interested in several metrics of a latency-sensitive TCP flow. We explore three metrics in this paper: 1) protocol latency distribution, and specifically, the percentage of packets that arrive at the receiver within a *delay threshold*, 2) average packet latency, and 3) normalized throughput, the ratio of the throughput of a MIN_BUF flow to a TCP flow. We choose two delay thresholds, 200 ms, which is related to interactive streaming performance, and 500 ms, which is somewhat arbitrary, but chosen to represent the requirements of responsive media streaming control operations.

In addition to comparing the latency behavior of standard TCP and MIN_BUF streams, we are also interested in understanding the effects on protocol latency of ECN enabled TCP.

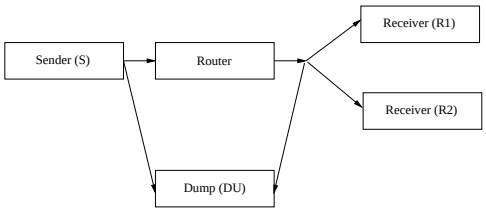


Fig. 1. Network Topology

Our results describe how this "streaming friendly" mechanism affects protocol latency.

B. Network Setup

All our experiments use a single-bottleneck "dumbbell" topology and FIFO scheduling at the bottleneck. The network topology is shown in Figure 1. Each box is a separate Linux machine. The latency and throughput measurements are performed for a single stream originating at the sender *S* and terminating at receiver *R1*. This stream is generated by an application that follows the application model described in Section II-C. The sender generates cross traffic for both receivers *R1* and *R2*. The router runs `nistnet` [20], a network emulation program that allows the introduction of additional delay and bandwidth constraints in the network path. The machine DU is used to dump TCP traffic for further analysis. The protocol latency is measured by recording the application write time for each packet on the sender *S* and the application read time for each packet on the receiver *R1*. All the machines are synchronized to within one ms of each other using NTP.

We chose three round-trip times (RTT) for the experiments and conducted separate experiments for each RTT. The RTTs were 25 ms, 50 ms and 100 ms. These RTTs approximate some commonly observed RTTs on the Internet. The cable modem from our home to work has 25 ms delay. West-coast to west-coast sites or East-coast to East-coast sites in the US observe 50 ms median delay and west-coast to east-coast sites in the US observe 100 ms median delay [9].

We run our experiments over standard TCP and ECN enabled TCP. For each RTT, two router queue lengths are chosen so that bandwidth is limited to 12 Mbs and 30 Mbs. The TCP experiments use tail dropping. For ECN, we use DRED active queue management [7], which is supported in `Nistnet`. DRED is a RED variant that is implemented efficiently in software. The *drdmin*, *drdmax* and *drdcongest* parameters of DRED were chosen to be 1.0, 2.0 and 2.0 times the bandwidth-delay product, respectively. DRED sends ECN messages for 10 percent of packets when the queue length exceeds *drdmin*, progressively increasing the percentage until packets are dropped when the queue length exceeds *drdcongest*. Unlike RED, DRED does not average queue lengths.

IV. Results

In this section, we discuss the results of our experiments. We start by showing the effects of using TCP and MIN_BUF streams on protocol latency. Then we quantify the throughput loss of these streams. We investigate the latencies observed at

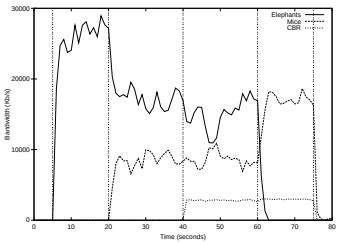


Fig. 2. The bandwidth profile of the cross traffic (15 elephants, 80 mice consuming about 30% bandwidth and 10% CBR traffic)

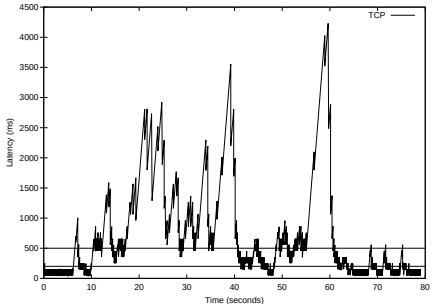
the sender, network and the receiver of TCP streams and the causes of each latency. Finally, we explore using ECN enabled TCP to improve protocol latencies.

A. Protocol Latency

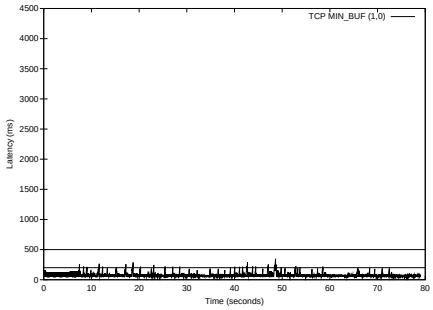
Our first experiment shows the protocol latency of TCP and MIN_BUF streams in response to dynamically changing network load. The experiment is run for about 80 seconds with load being introduced at various different time points in the experiment. The TCP or MIN_BUF long-lived stream being measured is started at t = 0s. We refer to this flow as the *latency* flow. Then at t = 5s, 15 other long-lived (*elephant*) flows are started, 7 going to receiver R1 and 8 going to receiver R2. At t = 20s, each receiver initiates 40 simultaneous short-lived (*mouse*) TCP flows. A mouse flow is a repeating short-lived flow that starts the connection, transfers 20KB of data, ends the connection and then repeats this process continuously [11]. The number of mouse flows was chosen so that the mouse flows would get approximately 30 percent of the total bandwidth. At t = 40s, CBR traffic that consumes 10 percent of the bandwidth is started. At t = 60s, the elephants are stopped and then the mice and the CBR traffic are stopped at t = 75s. Figure 2 shows the cross traffic (elephants, mice and CBR traffic) for a 30 Mbs bandwidth, 100 ms RTT experiment. Other experiments have a similar bandwidth profile.

Figure 3 shows the results of a run with standard TCP and MIN_BUF$(1, 0)$ streams when the bandwidth limit is 30Mbs and the round trip time is 100 ms. Both these streams originate at sender S and terminate at receiver R1. The figures shows the protocol latency of the latency flow as a function of packet receive time. The two horizontal lines on the *y* axis show the 200 ms and the 500 ms latency threshold.

Figure 4 shows the protocol latency of the three MIN_BUF configurations. Note that in this figure, the maximum value of the y axis is 500 ms. These figures show that the MIN_BUF streams have significantly lower protocol latency than a standard TCP stream. They show that, as expected, the MIN_BUF$(1, 0)$ flow has the lowest protocol latency while the MIN_BUF$(2, 0)$ has the highest protocol latency among the MIN_BUF flows. Looking at the throughput profile of the latencies flows (now shown here), we found that the protocol latency of TCP and MIN_BUF$(2, 0)$ is highest when the flow throughput is lowest. However, the protocol latency of MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows is not affected as much by their changing throughput. The reason is that the TCP send buffer drains slowly when the bandwidth available to the latency stream goes down. Since TCP and MIN_BUF$(2, 0)$ flows allow



(a) TCP



(b) MIN_BUF(1,0)

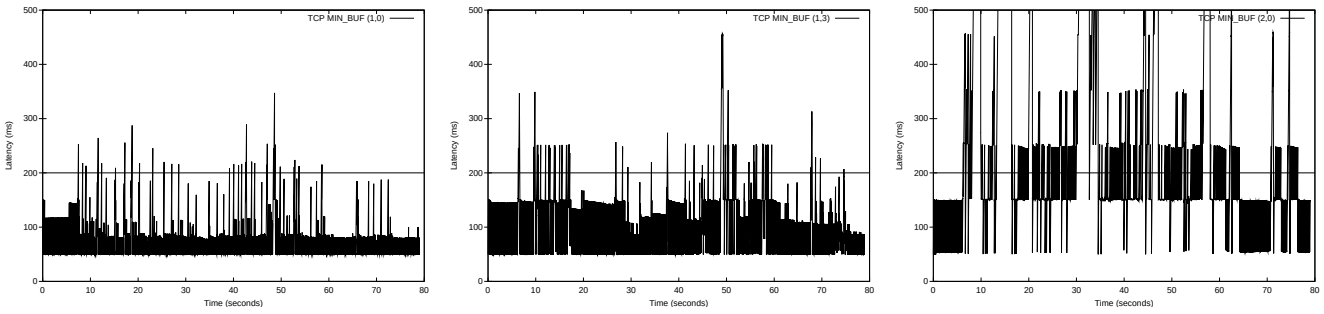
These figures show the protocol latency of packets plotted as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 200 ms and 500 ms latency threshold.

Fig. 3. A comparison of the protocol latencies of TCP and MIN_BUF(1,0) streams

the send buffer to fill up more than the other two flows, these flows observe higher protocol latencies. The send buffer does not significantly affect the protocol latency in MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows. The latency spikes seen in these flows are chiefly a result of TCP congestion control and retransmission as discussed in Section IV-C.

The protocol latency distribution for this experiment is shown in Figure 5. The experiment was performed with 30Mbs and 12Mbs bandwidth limit and with 100 ms, 50ms and 25 ms RTT. Each experiment was performed 8 times and the results presented show the numbers accumulated over all the runs. The vertical lines show the 200 and 500 ms delay thresholds. The figures show that in all cases a much larger percent of TCP packets lie outside the delay thresholds as compared to MIN_BUF flows. Note that the x axis, which shows the protocol latency in milliseconds, is on a log scale. The figures show that, as expected, the percent of packets with large delays increases with increasing RTT and decreasing bandwidth. The percent of packets delivered within the 200 and 500 ms delay thresholds is summarized in Table I. This table also shows that the packets delivered within the delay thresholds is very similar for MIN_BUF$(1, 0)$ and MIN_BUF$(1, 3)$ flows.

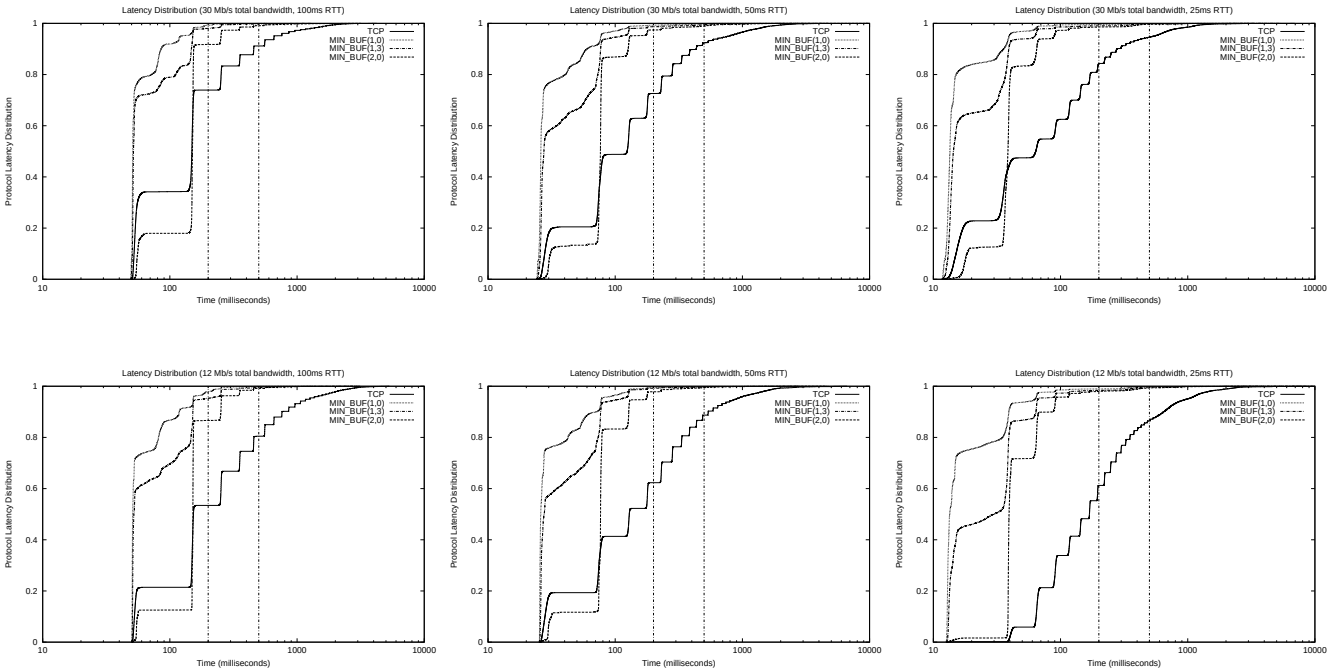
The average (one way) protocol latency for each configuration is shown in Table II. Each experiment was performed 8 times and these numbers are the mean of the 8 runs. The table



(a) MIN_BUF(1,0) (b) MIN_BUF(1,3) (c) MIN_BUF(2,0)

These experiments were performed under the same conditions as described in Figure 3. Note that the maximum value of the y axis is 500 ms, while it is 4500 ms in Figure 3.

Fig. 4. A comparison of the protocol latencies of 3 MIN_BUF configurations



The experiment was performed with a 30Mbs and 12Mbs bandwidth limit and with 100 ms, 50ms and 25 ms RTT. The vertical lines show the 200 and 500 ms delay thresholds. The x axis, which shows the protocol latency in milliseconds, is on a log scale.

Fig. 5. Protocol Latency Distribution of standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ flows

shows that MIN_BUF flows have much lower average latency and the deviation across runs is also much smaller.

B. Throughput Loss

We are interested in the throughput loss of MIN_BUF streams. We measured the throughput of each of the flows as a ratio of the total number of bytes received to the duration of the experiment. Table III shows the normalized throughput of each flow, which is the ratio of the throughput of the flow to the TCP flow. Again, these numbers are the mean (and 95% confidence interval) over 8 runs.

The table shows that the MIN_BUF$(2,0)$ flows receive throughput close to standard TCP (within the confidence range). MIN_BUF$(2,0)$ flows have CWND new packets that can be sent after a packet transmission. So even if all current CWND packets in flight are acknowledged almost simultaneously, TCP can send its entire next window of CWND packets immediately. Thus we expect that MIN_BUF(2,0) flows should behave similar to TCP flows.

The MIN_BUF$(1,0)$ flows consistently receive the least throughput, about 70 percent of TCP. This result is not surprising because TCP has no new packets in the send buffer that can be sent after each packet is transmitted. TCP must ask the application to write the next packet to the send buffer before it can proceed with the next transmission. Thus, any scheduling or other system delays would make the MIN_BUF$(1,0)$ flow an application-limited flow. TCP assumes that such flows need

		RTT = 100 ms		*RTT = 50 ms*		*RTT = 25 ms*	
Mbs	*Type*	*D200*	*D500*	*D200*	*D500*	*D200*	*D500*
30	std	0.73	0.91	0.72	0.92	0.84	0.94
30	m10	0.99	1.00	0.99	1.00	1.00	1.00
30	m13	0.98	1.00	0.99	0.99	0.99	1.00
30	m20	0.91	0.99	0.97	0.99	0.99	1.00
12	std	0.53	0.80	0.62	0.88	0.60	0.86
12	m10	0.98	1.00	0.99	1.00	0.99	1.00
12	m13	0.95	0.99	0.99	1.00	0.98	1.00
12	m20	0.86	0.99	0.97	0.99	0.98	0.99

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. The terms D200 and D500 refer to a delay threshold of 200 and 500 ms.

TABLE I

PERCENT OF PACKETS DELIVERED WITHIN 200 AND 500 MS THRESHOLDS FOR STANDARD TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ AND MIN_BUF$(2,0)$ FLOWS

Mbs	*Type*	*RTT = 100 ms*	*RTT = 50 ms*	*RTT = 25 ms*
30	std	226.31±0.87	218.84±40.34	138.61±21.0
30	m10	62.91±0.96	37.09±0.80	19.71±0.89
30	m13	76.19±2.71	51.54±3.73	28.29±1.70
30	m20	152.14±9.13	89.74±5.32	48.21±2.19
12	std	369.22±50.32	260.27±23.15	296.25±47.49
12	m10	69.73±2.15	38.50±1.09	25.94±1.80
12	m13	91.42±6.81	49.17±2.03	39.08±3.39
12	m20	162.26±6.06	87.90±1.46	61.31±5.59

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. All average latency numbers (together with 95% confidence intervals) are shown in milliseconds.

TABLE II

AVERAGE LATENCY OF STANDARD TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ AND MIN_BUF$(2,0)$ FLOWS

less bandwidth and reduces the window and thus the transmission rate of such flows.

Interestingly, the MIN_BUF$(1,3)$ flows receive throughput close to TCP, about 90 percent of TCP or more. Three additional packets in the send buffer (in addition to the CWND packets in flight) seem to reduce the throughput loss due to the artificial application-flow limitation introduced by MIN_BUF$(1,0)$ flows.

For a latency sensitive, quality-adaptive application, one metric for measuring the average flow quality could be the product of the percent of packets that arrive within a delay threshold and the normalized throughput of the flow. This relative metric is related to the number of packets that arrive within the delay threshold across different flows. Thus a larger value of this metric could imply better perceived quality. From the numbers presented above, MIN_BUF(1,3) flows have the highest value for this quality metric because both their delay threshold numbers (shown in Table I) and normalized throughput numbers (shown in Table III) are close to the best numbers of other flows.

Mbs	*Type*	*RTT = 100 ms*	*RTT = 50 ms*	*RTT = 25 ms*
30	std	1.00	1.00	1.00
30	m10	0.66±0.11	0.71±0.08	0.76±0.10
30	m13	0.96±0.12	0.87±0.08	0.92±0.12
30	m20	1.02±0.18	1.13± 0.36	0.91±0.10
12	std	1.00	1.00	1.00
12	m10	0.67±0.09	0.76±0.05	0.89±0.11
12	m13	0.92±0.15	1.06±0.09	1.08±0.22
12	m20	1.13±0.16	1.08±0.14	1.12±0.17

The terms `std, m10, m13` and `m20` refer to standard TCP, MIN_BUF$(1,0)$, MIN_BUF$(1,3)$ and MIN_BUF$(2,0)$ respectively. The normalized throughput (NT) is the ratio of throughput of each flow to the ratio of a standard TCP flow.

TABLE III

THE NORMALIZED THROUGHPUT OF A STANDARD TCP FLOW AND MIN_BUF FLOWS

C. Understanding Worst Case Behavior

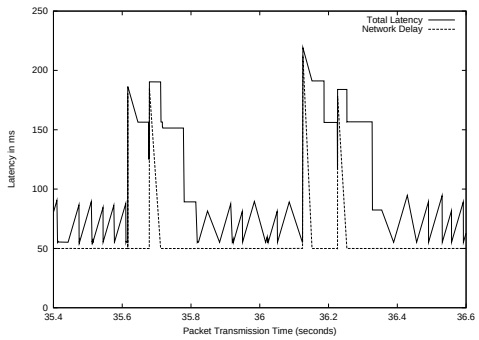
Figure 4 shows that MIN_BUF$(1,0)$ and MIN_BUF$(1,3)$ flows occasionally show protocol latency spikes even though they have small send buffers. To understand the cause of these spikes, we measured the delays experienced by each packet on the sender side, in the network and on the receiver side.

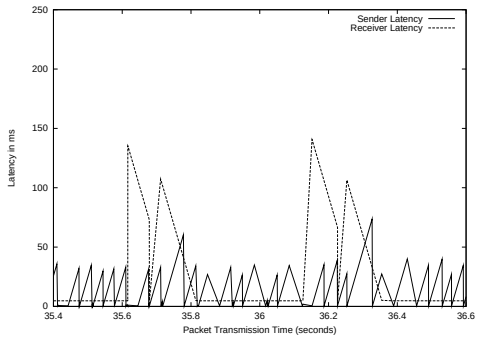
Figure 6 shows these delays for a small part of the experiment when packets were lost and retransmitted. The sender latency of each packet is the time from when an application writes to the socket to TCP's first transmission of the packet. The network delay is the time from the first transmission of each packet to the first arrival at the receiver. The receiver latency is the time from the first arrival of each packet to an application read. Figure 6 shows that the latency spikes are primarily caused by packet losses and retransmissions. In particular, the protocol (or total) latency does not depend significantly on the flow throughput (or the congestion window size). For instance, the congestion window size at t=35.5 ms and t=36.5 ms is 15 and 4, but the total latency at these times is roughly the same.

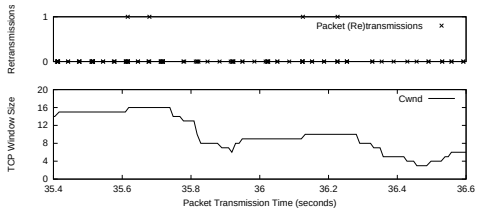
Packet retransmissions initially cause the network delay to increase, followed by an increase in the receiver latency. The receiver latency increases because TCP delivers packets in order and a lost packet temporarily blocks further packets from being released to the application. In addition, the sender latency increases slightly because TCP reduces its congestion window after a packet loss. Thus packets that were already accepted into the send buffer are delayed. Note that after a packet loss, increases in latency at the network, receiver and the sender are typically not additive (for any given packet) since they are shifted in time. However, this time shifting implies that the total latency stays high for several packets after a packet is dropped. These findings motivated the need to explore mechanisms that can reduce packet dropping. One such mechanism that has been studied by the networking community is explicit congestion notification (ECN) [24], [28].

D. Protocol Latency with ECN

ECN enabled routers inform TCP senders of impending congestion by setting an ECN bit on certain packets. When an ECN enabled TCP sender receives such a packet, it takes pro-active







This experiment was performed with a MIN_BUF$(1, 0)$ flow. The bandwidth limit is 30 Mbs and the RTT is 100 ms. All figures are plotted as a function of the packet transmission time. These figures show that the sender side latency is small for MIN_BUF$(1, 0)$ flows and that spikes in total latency occur primarily due to packet loss and retransmissions.

Fig. 6. The packet delay on the sender side, the network and the receiver side

measures to reduce its sending rate to avoid packet dropping in the router.

We ran the same set of experiments as described in Section IV-A to measure and compare the protocol latency of ECN flows and MIN_BUF (with ECN) flows. Figure 7 shows the bandwidth profile of the competing traffic. Figures 8 and 9 show the comparative protocol latencies. These figures are generated from experiments that are similar to those shown in Figure 3 except we enabled ECN at the end points and used DRED active queue management at the intermediate router.

These figures show that the protocol latency spikes are reduced in all cases when compared to Figure 4. A close look at the raw data showed that ECN reduced packet dropping and retransmissions and thus had fewer spikes. More experimental results for ECN can be found in an extended version of this paper [8].

ECN in these experiments showed several interesting band-

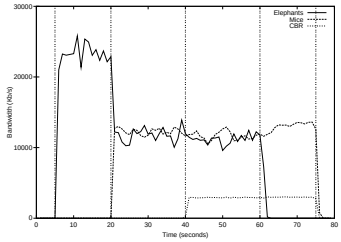
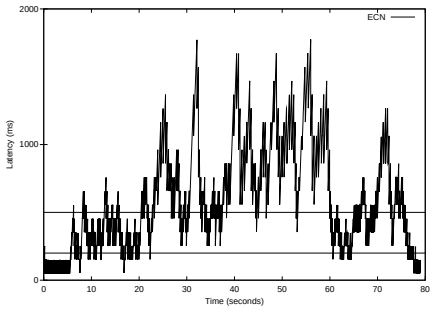
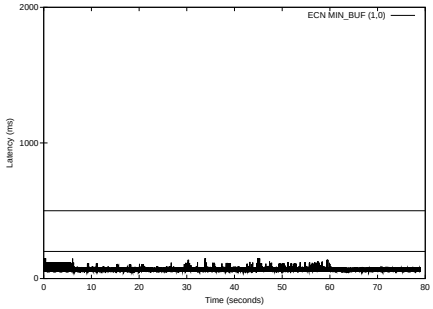


Fig. 7. The bandwidth profile of the cross traffic (15 elephants, 80 mice consuming about 50% bandwidth and 10% CBR traffic)



(a) ECN

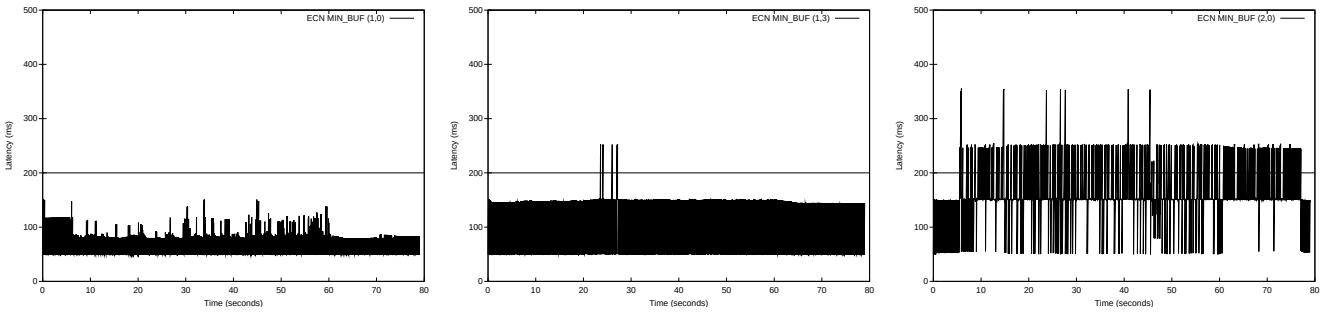


(b) MIN_BUF(1,0) with ECN

These figures show the protocol latency as a function of packet receive time. The bandwidth limit for this experiment is 30 Mbs and the round trip time is 100 ms. The horizontal lines on the figures show the 200 ms and 500 ms latency threshold.

Fig. 8. A comparison of the protocol latencies for ECN and MIN_BUF$(1, 0)$ streams

width related properties. First, the mouse bandwidth was tuned to 50 percent of the bandwidth capacity as shown in Figure 7, instead of 30 percent as shown in Figure 2. The mice were able to achieve their bandwidth share quickly and more accurately. With TCP, in some configurations (lower bandwidth and smaller RTT), the mice were not able to achieve 50 percent bandwidth share even when the application starts very large numbers of mice. This is because the elephants are very aggressive and the mouse are unable to connect for long periods of time. In addition, the ratio of mice to elephants needed to achieve fair sharing between the mice and the elephants is much smaller for ECN than with regular TCP flows. Thus, elephants do not steal as much bandwidth from mice and also have a smoother throughput profile (not shown here). We believe that



(a) MIN_BUF(1,0) (b) MIN_BUF(1,3) (c) MIN_BUF(2,0)

These experiments were performed under the same conditions as described in Figure 8. Note that the maximum value of the y axis is 500 ms, while it is 2000 ms in Figure 8.

Fig. 9. A comparison of the protocol latencies of 3 MIN_BUF configurations

although ECN may loose throughput compared to TCP for long lived flows, its reduced aggressiveness leads to fewer retransmissions and thus it is desirable for low latency streaming.

V. Related Work

The feasibility of TCP-based stored media streaming has been studied by several researchers. Generally, the tradeoff in these QoS adaptive approaches is short-term improvement in video quality versus long term smoothing of quality. Rejaie [26] uses layered video and adds or drops video stream layers to perform long-term coarse grained adaptation, while using a TCP-friendly congestion control mechanism to react to congestion on short-time scales. Krasic [14] contends that new compression practices and reduced storage costs make TCP a viable and attractive basis for streaming stored content and uses standard TCP, instead of a TCP-friendly scheme, for media streaming. Feng [4] and Krasic use priority-based streaming, which allows a simpler and more flexible implementation of QoS adaptation. We believe that similar QoS adaptive approaches will be useful for low latency streaming also.

Researchers in the multimedia and networking community have proposed several alternatives to TCP for media streaming [30], [6]. These alternatives aim to provide TCP-friendly congestion control for media streams without providing reliable data delivery and thus avoid the latency introduced by packet retransmissions. Unfortunately, the effects of packet loss on media streaming are non-uniform and can quickly become severe. For instance, loss of the header bits of an *I*-frame in an MPEG movie can render a large segment of surrounding video data unviewable. Thus media applications over a lossy transport protocol have to implement complex recovery strategies such as FEC [27] that potentially have high bandwidth and processing overhead. The benefit of FEC schemes for loss recovery is that they often have lower latency overhead as compared to ARQ schemes such as employed in TCP. Thus, Nonnenmacher [21] explores introducing FEC as a transparent layer under an ARQ scheme to improve transmission efficiency.

Popular interactive streaming applications include Voice over IP (VoIP) products such as Microsoft NetMeeting [19]. NetMeeting provides reasonable voice quality over a best effort network but is implemented over UDP because the delays introduced by TCP are considered unacceptable. This paper shows that MIN_BUF TCP should yield acceptable delays, especially for QoS adaptive applications. For interactive applications, ITU G.114 [12] recommends 150 ms as the upper limit for one-way delay for most applications, 150 to 400 ms as potentially tolerable, and above 400 ms as generally unacceptable delay. The one way delay tolerance for video conferencing is in a similar range, 200 to 300 ms.

Our send-buffer adaptation approach is similar to the buffer tuning work by Semke [29]. Semke tunes the send buffer size to between $2*$CWND and $4*$CWND to improve the throughput of a high bandwidth-delay connection that is otherwise limited by the send buffer size. The $4*$CWND value is chosen to limit small, periodic fluctuations in buffer size. This paper shows that a connection can achieve throughput close to TCP throughput by keeping the send buffer size slightly larger than CWND and also achieve significant reduction in protocol latency.

Many differentiated network services have been proposed for low latency streaming. These schemes are complementary to our work since, generally, a MIN_BUF TCP implementation can be used for the low delay flow. Hurley [10] provides a low-delay alternative best-effort (ABE) service that trades high throughput for low delay. The ABE service drops packets in the network if the packets are delayed beyond their delay constraint. In this model, the client must recover from randomly dropped packets. Further, unlike with TCP, the server does not easily get back-pressure feedback information from the network in order to make informed QoS adaptation decisions.

Active queue management and explicit congestion notification (ECN) [24] have been proposed for improving the packet loss rates of TCP flows. Salim [28] shows ECN has increasing throughput advantage with increasing congestion levels and ECN flows have hardly any retransmissions. Feng [3] shows that adaptive active queue management algorithms (Adaptive RED) and more conservative end-host mechanisms can significantly reduce loss rates across congested links.

Claffy [1] presents the results of a measurement study of the T1 NSFNET backbone and delay statistics. In 1992, the one way median delays between end points ranges from 20 to 80 ms with a peak at 45 ms. Newer data in 2001 [9] shows that

the median RTT for East-coast to East-coast or West-coast to West-coast is 25-50 ms and East-coast to West-coast is about 100 ms. We use these median results in our experiments. US to Europe median RTT is currently 200 ms. While the 200 ms median RTT makes interactive applications challenging, responsive control operations for streaming media should be possible.

VI. Future Work

The results in this paper are based on experiments conducted over an experimental network test-bed. While simulating our experiments under more exhaustive conditions using a network simulator, such as `ns`, would be useful, the task is not trivial because `ns` does not simulate the send buffer. Thus a simulator for the send buffer would have to be implemented. In addition, we are interested in observing whether scheduling and other timing effects change the latency or throughput behavior of MIN_BUF streams. Simulating such effects is beyond the scope of `ns`.

We have explored adapting the send buffer using three different sizes for MIN_BUF(A, B) flows. These different configurations, with increasing buffer sizes, have increasing latency and throughput. Another approach for adapting the send buffer is to auto-tune the values of A and B so that the send buffer contributes a certain amount of delay while providing the best possible throughput.

We are currently implementing a streaming quality-adaptive media server that will allow channel surfing as well as basic control operations such as fast forward, stop, rewind, etc. We plan to compare the latency of these operations using standard TCP versus MIN_BUF flows. We are also integrating a real-time MPEG encoder into the media server, which will allow us to investigate some of the challenges raised by low latency streaming, including the handling of late packets.

VII. Conclusions

The dominance of the TCP protocol on the Internet and its success in maintaining Internet stability has led to several TCP-based stored media-streaming approaches. These approaches use a combination of client-side buffering and QoS adaptation to overcome various problems that were considered inherent with TCP-based media streaming.

The success of TCP-based streaming led us to explore the limits to which TCP can be used for low-latency media streaming. Low latency streaming allows responsive streaming control operations and sufficiently low latency streaming would make interactive applications feasible. We examined adapting the TCP send buffer size based on TCP's congestion window to reduce protocol latency or application perceived network latency. Our results show that this simple idea reduces protocol latency and significantly improves the number of packets that can be delivered within 200 ms and 500 ms thresholds.

References

[1] Kimberly C. Claffy, George C. Polyzos, and Hans-Werner Braun. Traffic Characteristics of the T1 NSFNET Backbone. In *INFOCOM*, pages 885–892, 1993.

[2] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 200–208, Philadelphia, PA, 1990.

[3] Wu-chang Feng, Dilip D. Kandlur, Debanjan Saha, and Kang S. Shin. Techniques for Eliminating Packet Loss in Congested TCP/IP Networks. Technical Report CSE-TR-349-97, U. Michigan, Nov 1997.

[4] Wu-Chi Feng, Ming Liu, Brijesh Krishnaswami, and Arvind Prabhudev. A Priority-Based Technique for the Best-Effort Delivery of Stored Video. In *Proc. of SPIE Multimedia Computing and Networking Conference (MMCN)*, January 1999.

[5] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[6] Sally Floyd, Mark Handley, and Eddie Kohler. Problem Statement for DCP. Work in progress, IETF Internet Draft draft-floyd-dcp-problem-00.txt, expires Aug 2002, Feb 2002.

[7] M. Gaynor. Proactive Packet Dropping Methods for TCP Gateways. `http://www.eecs.harvard.edu/~gaynor/final.ps`, October 1996.

[8] Ashvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting Low Latency TCP-Based Media Streams. Technical Report CSE-02-002, Oregon Graduate Institute, March 2002. `ftp://cse.ogi.edu/pub/tech-reports/2002/02-002.ps.gz`.

[9] Bradley Huffaker, Marina Fomenkov, David Moore, and kc claffy. Macroscopic Analyses of the Infrastructure: Measurement and Visualization of Internet Connectivity and Performance. In *A workshop on Passive and Active Measurements*, Amsterdam, April 2001.

[10] P. Hurley and J. Y. Le Boudec. A Proposal for an Asymmetric Best-Effort Service. In *Proceedings of IEEE/IFIP IWQoS 1999*, pages 132–134, May 1999.

[11] Gianluca Iannaccone, Martin May, and Christophe Diot. Aggregate Traffic Performance with Active Queue Management and Drop from Tail. *ACM Computer Communication Review*, 31(3), July 2001.

[12] International Telecommunication Union (ITU). *Transmission Systems and Media, General Recommendation on the Transmission Quality for an Entire International Telephone Connection; One-Way Transmission Time*. Geneva, Switzerland, March 1993. Recommendation G.114, Telecommunication Standardization Sector of ITU.

[13] V. Jacobson. Congestion Avoidance and Control. In *ACM SIGCOMM*, pages 314–329, Stanford, CA, August 1988.

[14] Charles Krasic, Kang Li, and Jonathan Walpole. The Case for Streaming Multimedia with TCP. In *8th International Workshop on Interactive Distributed Multimedia Systems (iDMS 2001)*, pages 213–218, Sep 2001. Lancaster, UK.

[15] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. Internet RFC 2018, October 1996.

[16] Matthew Mathis and Jamshid Mahdavi. Forward Acknowledgment: Refining TCP Congestion Control. In *ACM SIGCOMM*, 1996.

[17] J. McCann, S. Deering, and J. Mogul. Path MTU Discovery for IP version 6. Internet RFC 1981, August 1996.

[18] Microsoft Inc. Windows Media Player. `http://www.microsoft.com/windows/windowsmedia`.

[19] Microsoft Inc. Windows NetMeeting. `http://www.microsoft.com/netmeeting`.

[20] NIST. The NIST Network Emulation Tool. `http://www.antd.nist.gov/itg/nistnet`.

[21] Jörg Nonnenmacher, Ernst W. Biersack, and Don Towsley. Parity-Based Loss Recovery for Reliable Multicast Transmission. *ACM/IEEE Transactions on Networking*, 6(4):349–361, 1998.

[22] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the 1999 USENIX Technical Conference*, pages 199–212, Monterey, CA, June 1999.

[23] V. Paxson. End-to-End Internet Packet Dynamics. In *ACM SIGCOMM*, pages 139–152, September 1997.

[24] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. Internet RFC 3168, September 2001.

[25] Real Networks. RealPlayer Streaming Media Player. `http://www.real.com`.

[26] Reza Rejaie, Mark Handley, and Deborah Estrin. Quality Adaptation for Congestion Controlled Video Playback over the Internet. In *ACM SIGCOMM*, pages 189–200, 1999.

[27] Luigi Rizzo. Effective Erasure Codes for Reliable Computer Communication Protocols. *ACM Computer Communication Review*, 27, 1997.

[28] J. Hadi Salim and U. Almed. Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks. Internet RFC 2884, July 2000.

[29] Jeffrey Semke, Jamshid Mahdavi, and Matthew Mathis. Automatic TCP Buffer Tuning. In *ACM SIGCOMM*, pages 315–323, 1998.

[30] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. Internet RFC 2960, Oct 2000.

APPENDIX I

Infopipes: An Abstraction for Multimedia Streaming. Andrew Black, Rainer Koster, Jie Huang, Jonathan Walpole, and Calton Pu, Multimedia Systems (special issue on Multimedia Middleware), 8(5), pp. 406-419, ACM / Springer-Verlag, 2002.

Infopipes: an Abstraction for Multimedia Streaming

Andrew P. Black, Jie Huang, Rainer Koster,
Jonathan Walpole and Calton Pu

Department of Computer Science and Engineering
OGI School of Science & Engineering
Oregon Health & Science University
20000 NW Walker Road
Beaverton, OR 97006-8921 USA

Technical Report Number CSE 02-001
Revised: 17th April 2002

This paper has been accepted for publication in the ACM/Springer-Verlag *Multimedia Systems Journal*. It will appear in a special issue on multimedia middleware that is scheduled for publication in late 2002. References in the literature should be to *Multimedia Systems Journal.*

Multimedia Systems (2002) 0:000–000

Infopipes: an Abstraction for Multimedia Streaming

Andrew P. Black[1], Jie Huang[1], Rainer Koster[2], Jonathan Walpole[1], Calton Pu[3]

1. Department of Computer Science & Engineering, OGI School of Science & Engineering, Oregon Health & Science University
2. Fachbereich Informatik, University of Kaiserslautern
3. School of Computing, Georgia Institute of Technology.

Abstract. To simplify the task of building distributed streaming applications, we propose a new abstraction for information flow—Infopipes. Infopipes make information flow primary, not an auxiliary mechanism that is hidden away. Systems are built by connecting pre-defined component Infopipes such as sources, sinks, buffers, filters, broadcasting pipes, and multiplexing pipes. The goal of Infopipes is not to hide communication, like an RPC system, but to *reify* it: to represent communication explicitly as objects that the program can interrogate and manipulate. Moreover, these objects represent communication in application-level terms, not in terms of network or process implementation.

Key words: Quality of service — Streaming — Communication — Feedback — Real-rate systems

1 Introduction

Recent years have witnessed a revolution in the way people use computers. In today's Internet-dominated computing environment, information exchange has replaced computation as the primary activity of most computers. This revolution began with the use of the world wide web for accessing relatively static information. It has continued with the emergence of streaming applications, such as video and music on demand, IP-telephony, Internet radio, video conferencing and remote surveillance systems. Recent traffic studies (Chesire et al. 2001; Thompson et al. 1997) show that these applications are already major consumers of bandwidth on the Internet and are likely to become dominant in the near future. The advent of interconnected, embedded and sensor-based systems will accelerate the development and deployment of new streaming applications.

The salient characteristics of streaming applications are their extensive use of communication among distributed components and their real-time interaction with real-world processes. Consequently, developers of streaming applications spend much of their time reasoning about the communications and I/O behaviour of their systems. This change of emphasis from computation to communication is the motivation for our research.

This paper describes *Infopipes*, a new abstraction, together with associated middleware and tools, for simplifying the task of constructing streaming applications. The motivation for developing Infopipes as middleware is to provide a single set of abstractions with a uniform interface that can be made available on a diverse set of hosts and devices. Wide availability is important for streaming applications because, by their nature, they tend to span many potentially heterogeneous computers and networks, and interact with many different devices. The abstractions must also be appropriate to the problem domain: they should expose the primitives useful in that domain, and control and hide the unnecessary details.

The essence of streaming applications is creation and management of information flows via producer-consumer interactions among potentially distributed components. Hence, communication is a primary concern and should be *exposed,* not hidden. Moreover, it is application level information that must be communicated, not low-level data, so exposing low-level network abstractions is inappropriate.

Exposing the basic communication elements, such as sources, sinks and routes, is inadequate: streaming applications are frequently also concerned with the quality of service (QoS) of that communication. For example, the correct execution of a streaming media application is often critically dependent on the available bandwidth between the server and client. Adaptive applications may actively monitor this QoS aspect and adapt the media quality dynamically to match their bandwidth requirements to the available bandwidth (Jacobs and Eleftheriadis 1998; Karr et al. 2001; McCanne et al. 1997; Walpole et al. 1997). When constructing streaming applications these timing and resource management-related tasks tend to be the source of much of the complexity, since they touch on aspects of the environment that differ among applications, and even among different deployments of the same application. In contrast, the computation-intensive aspects of the application, such as media encoding and

Correspondence to: A.P.Black

decoding, can often be addressed using standard components.

The desire to reify communication rather than hide it is in contrast to many distributed systems middleware platforms that are based around remote procedure call (RPC) mechanisms (ISO 1998; OMG 1998b; OSF 1991; Sun 2002). Of course, it is also undesirable and unmanageable to expose all of the underlying details of communication. In general, information-flow application developers will not want to reimplement low-level protocol functionality — such as marshalling, fragmentation, congestion control, ordered delivery and reliability — for every application they build. Thus, what we would like to do is to find a way to factor and pre-package such functionality so that it can be selected when needed to ensure a particular property. This emphasis on component-based composition and property composition is a central characteristic of the Infopipe approach.

This approach to dealing with the complexities of communication can be re-applied when dealing with the complexities of scheduling computation. Since timing is critical for many streaming applications, they need some way to control it. However, it is neither desirable nor necessary to expose application developers to all of the ugly details of thread management, scheduling, and synchronization. Instead we attempt to expose the QoS-related aspects of scheduling and hide the unnecessary details.

Thus, our primary goal for Infopipes is to select a suitable set of abstractions for the domain of streaming applications, make them available over a wide range of hardware and operating systems, and allow tight control over the properties that are important in this domain while hiding the unnecessary details.

A further goal, which we discovered to be important through our own experiences building real-time streaming applications, is the ability to monitor and control properties dynamically and in application-specific terms. This capability enables applications to degrade or upgrade their behaviour *gracefully* in the presence of fluctuations in available resource capacity. Since graceful adaptation is an application-defined concept it cannot be achieved using a one-size-fits-all approach embedded in the underlying systems software. The alternate approach of exposing system-level resource-management information to application developers introduces unnecessary complexity into the task of building applications. Therefore, the goal for a middleware solution is to map system-level resource management details into application-level concepts so that adaptive resource management can be performed by application components in application-specific terms.

A final goal for Infopipe middleware is to support tools that automatically check the properties of a composite system. For example, important correctness properties for a pipeline in a streaming application are that information be able to flow from the source to the sink, that latency bounds are not exceeded and that the quality of the information meets the requirements. Even though individual Infopipe components may exhibit the necessary properties in isolation, it is often non-trivial to derive the properties of a system that is composed from these components.

The remainder of this paper presents more detail about our ongoing research on Infopipes. Section 2 discusses the Infopipe model, loosely based on a plumbing analogy, and describes the behaviour of various basic Infopipe components. Section 3 discusses some of the properties that are important for composite Infopipes and introduces some preliminary tools we have developed. Section 4 discusses the implementation. Some example Infopipe applications are presented in section 5. Section 6 discusses related work, and section 7 concludes the paper.

2 The Infopipe Model and Component Library

Infopipes are both a model for describing and reasoning about information-flow applications, and a realization of that model in terms of objects that we call Infopipe components. It is central to our approach that these components are real objects that can be created, named, configured, connected and interrogated at will; they exist at the same level of abstraction as the logic of the application, and this expose the application-specific information flows to the application in its own terms. For example, an application object might send a method invocation to an Infopipe asking how many frames have passed through it in a given time interval, or it might invoke a method of an Infopipe that will connect it to a second Infopipe passed as an argument.

An analogy with plumbing captures our vision: just as a water distribution system is built by connecting together pre-existing pipes, tees, valves and application-specific fixtures, so an information flow system is built by connecting together pre-defined and application-specific Infopipes. Moreover, we see Infopipes as a useful tool for modelling not only the communication of information from place to place, but also the transformation and filtering of that information. The Infopipe component library therefore includes processing and control Infopipes as well as communication Infopipes. We can also compose more complex Infopipes with hybrid functionality from these basic components. Our goal is to provide a rich enough set of components that we can construct information flow networks, which we call Infopipelines, for a wide variety of applications.

2.1 Anatomy of an Infopipe

Information flows into and out of an Infopipe through *Ports*; *push* and *pull* operations on these ports constitute the Infopipe's data interface. An Infopipe also has a control interface that allows dynamic monitoring and control of its properties, and hence the properties of the information flowing through it. Infopipes also support connection interfaces that allow them to be composed, *i.e.*, connected together at run-time, to form Infopipelines. The major interfaces required for an object to be an Infopipe are shown in Fig. 1.

It is central to our approach that Infopipes are *compositional*. By this we mean that the properties of a pipeline can be calculated from the properties of its individual Infopipe components. For example, if an Infopipe with a latency of 1 ms is connected in series with an Infopipe with a latency of 2 ms, the resulting pipeline should have a latency of 3 ms — not 3.5 ms or 10 ms.

Compositionality requires that connections between components are *seamless*: the cost of the connection itself must be insignificant. Pragmatically, we treat a single procedure

Cloning	
clone	answers a disconnected copy of this Infopipe
Data	
pull	answers an item obtained from this Infopipe.
push: anItem	push an item into this pipe
Connection	
--> aPortOrInfopipe	connect my Primary outport to aPortOrInfopipe
Port Access	
inPort	answers my Primary Inport
inPortAt: name	answers my named inport
inPorts	answers a collection containg all of my inports
outPort	answers my primary Outport
outPortAt: name	answers my named Outport
outPorts	answers a collection containg all of my outports
nameOfInPort: anInPort	answers the name of anInPort
nameOfOutPort: anOutPort	answers the name of anOutPort
openInPorts	answers a collection containing all of my Inports that are not connected
openOutPorts	answers a collection containing all of my Outports that are not connected
Pipeline Access	
allConnectedInfoPipes	answers a collection containing all of the Infopipes in the same Infopipeline as myself.
inConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Inports
outConnectedTo	answers a collection containing all of the Infopipes that are directly connected to my Outports

call or method invocation as having insignificant cost. In contrast, a remote procedure call, or a method that might block the invoker, have potentially large costs: we do not allow such costs to be introduced automatically when two Infopipes are connected. Instead, we encapsulate remote communication and flow synchronization as Infopipe components, and require that the client include these components explicitly in the Infopipeline. In this way the costs that they represent can also be included explicitly.

Other properties may not compose so simply as latency. For example, CPU load may not be additive: memory locality effects can cause either positive or negative interference between two filters that massage the same data. While we do not yet have solutions to all of the problems of interference, we do feel strongly that addressing these problems requires us to be explicit about all of the stages in an information flow.

2.2 *Control Interfaces*

The control interface of an Infopipe exposes and manages two sets of properties: the properties of the Infopipe itself, and the properties of the information flowing through it. To see the distinction, consider an Infopipe implemented over a dedicated network connection. The bandwidth of this Netpipe is a property of the underlying network connection. However, the actual data flow rate, although bounded by the bandwidth, may vary with the demands of the application.

We regard both pipe and flow properties as control properties because they are clearly related. Indeed, expressing pipe properties such as bandwidth in application-level terms (*e.g.*, frames per second rather than bytes per second) requires information about the flow.

Different kinds of Infopipe provide different control interfaces. For example, we have `fillLevel` for buffers and `slower` and `faster` for pumps. We are investigating the properties and control information that should be maintained in Infopipes and in information flows to support comprehensive control interfaces.

2.3 *Ports*

To be useful as a component in an Infopipeline, an Infopipe must have at least one *port*. Ports are the means by which information flows from one Infopipe to another, and are categorized by the direction of information flow as either *Inports* (into which information flows, indicated by the symbol ▰) or *Outports* (from which information flows, indicated by the symbol ▰).

Each Infopipe has a set of named `Inports` and a set of named `Outports`; each port is owned by exactly one Infopipe. For straight-line pipes, both the Inport set and the Outport set have a single element, which is named `Primary`.

`OutPorts` have a method `–>> anInPort` that sets up a connection to `anInPort`. `Infopipes` also have a `–>>` method, which is defined as connecting the primary `OutPort` of the upstream pipe to the primary `InPort` of the downstream pipe.

Information can be passed from one Infopipe to another

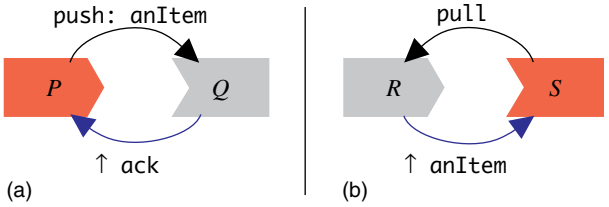


Fig. 2 . (a) Illustrates push mode communication; (b) illustrates pull mode communication

```
"Create some Infopipes"
source := SequentialSource new.
pump := Pump new.
multicastTee := MulticastTee new.
mixTee := MixTee new.
sink := Sink new.

"Connect them"
source ->> pump ->> multicastTee.
(multicastTee outPortAt: #Primary) ->> (mixTee inPortAt: #Primary).
(multicastTee outPortAt: #Secondary) ->> (mixTee inPortAt: #Secondary).
mixTee ->> sink.

"Make data items flow."
pump startPumping: 1000.

"result pipeline"
```

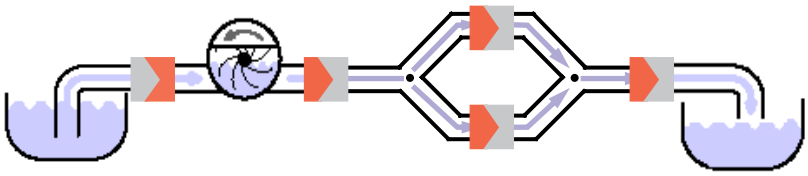


Fig. 4 : Building a pipeline with Tees

in two ways. In *push mode*, the Outport of the upstream component invokes the method `push: anItem`[1] on the Inport of the downstream component, as shown in Fig. 2(a). In *pull mode,* shown in Fig. 2(b), the situation is dual: the Inport of the downstream component invokes the `pull` method of the Outport of the upstream component, which replies with the information item. The ports *P* and *S* (coloured ■ in the figure) invoke methods; we say that they are *positive*. Ports *Q* and *R* (coloured ■) execute methods when invoked; we say that they are *negative*. In a well-formed pipeline, connected ports have opposite direction and opposite polarity. Any attempt to connect, for example, an Inport to another Inport, or a positive port to another positive port, should be rejected.

It is not obvious that Infopipes need the concept of port. Indeed, our first prototypes of "straight line" Infopipes did not have ports: a pipe was connected directly to its upstream and downstream neighbours, and each pipe had two connection methods, `input:` and `output:` However, the introduction of *Tees*, that is, pipes with multiple inputs and outputs, would have made the connection protocol more complex and less uniform. Ports avoid this complexity, and turn out to be useful in building `RemotePipes` and `CompositePipes` as well, as we shall explain later.

2.4 Common Components

Fig. 3 illustrates some Infopipe components. *Sources* are Infopipes in which the set of Inports is empty; whereas *Sinks* have an empty set of Outports. *Tees* are Infopipes in which one or both of these sets have multiple members. These ports can be accessed by sending the Tee the messages `inPortAt: aName` and `outPortAt: aName`; the ports can then be connected as required. Figure 4 shows an example. In addition, we can identify various other Infopipes.

- A *buffer* is an Infopipe with a negative Inport, a negative Outport, and some storage. The control interface of the buffer allows us to determine how much storage it should provide, and to ascertain what fraction is in use.
- A *pump* is an Infopipe with a positive Inport and a positive Outport. Its control interface lets us set the rate at which the pump should consume and emit information items.

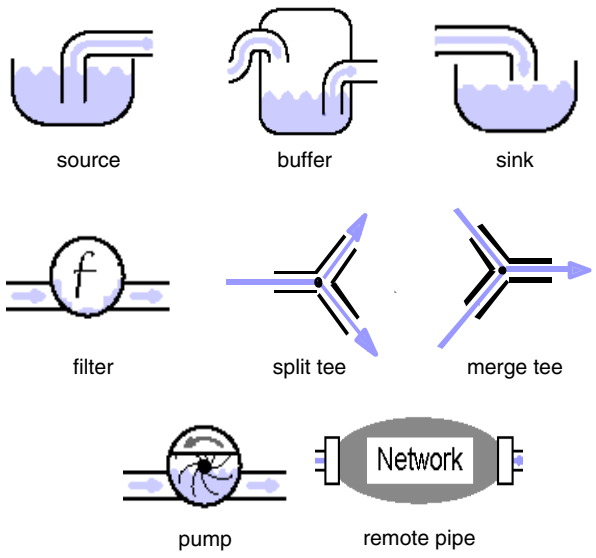


Fig. 3 . Some Infopipe components

1. We follow the Smalltalk convention of using a colon (rather than parenthesis) to indicate where an argument is required. Often, as here, we will provide an example argument with a meaningful name.

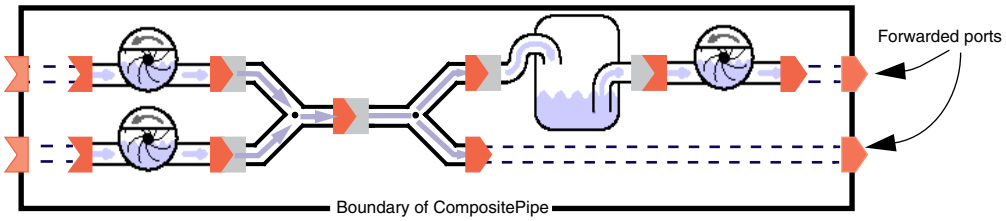


Fig. 5 : Internal Structure of a CompositePipe

– A *remote pipe* is an Infopipe that transports information from one address space to another. Although the Infopipe abstraction is at a higher level than that of address space, a middleware implementation must recognize that a host program executes in an address space that is likely to encompass only part of the Infopipeline. Remote pipes bridge this gap; the Inport and Outport of a remote pipe exist in different address spaces, and the remote pipe itself provides an information portal between those address spaces.

  Remote pipes can be constructed with different polarities, reflecting the different kinds of communication path. An IPCPipe between two address spaces on the same machine might provide reliable, low-latency, communication between a negative Inport and a positive Outport; such a pipe emits items as they arrive from the other address space. A Netpipe that connects two address spaces on different machines has two negative ports and provides buffering; items are kept until they are requested by the next connected Infopipe in the downstream address space.

An important aspect of component-based systems is the ability to create new components by aggregating old ones, and then to use the new components as if they were primitive. *Composite* pipes provide this functionality; any connected sub-network of Infopipes can be converted into a `CompositePipe`, which clients can treat as a new primitive.

In order for clients to connect to a composite pipe in the same way as to a primitive Infopipe, without knowing anything about its internal structure, and indeed without knowing that it *is* a composite rather than a primitive, a composite pipe must have its own ports. We call these ports `ForwardedPorts`. The `ForwardedPorts` are in one-to-one correspondence with, but are distinct from, the open ports of the sub-components. We cannot use the same object for the `ForwardedPort` and the real port because the real port is owned by the sub-component while the ForwardedPort is owned by the `CompositePipe` itself. Figure 5 shows the internal structure of a composite pipe. From the outside, it is just an ordinary Infopipe with two Inports and two Outports. Open ports of different sub-components may have the same name, but their `ForwardedPorts` must have different names because the ports of an Infopipe must be distinguishable.

One inevitable difference between composite and primitive Infopipes is that the former need more complex initialization: the internal structure of the Composite must be established before it can be used. It is therefore convenient to adopt a prototype-oriented style of programming, where a Composite is first constructed and then *cloned* to create as many instances as required. To support this style uniformly, all Infopipes (not just composite pipes) have a `clone` method, which makes a pipe-specific set of choices about what parameters to copy and what parameters to re-initialize. For example, when a Pump is cloned, the pumping rate is copied from the prototype, but the ports of the clone are left open.

3 From Pipes to Pipelines: Analysis & Tools

3.1 Polarity Checking and Polymorphism

The concept of port polarity introduced in section 2.3 is the basis for several useful checks that an Infopipeline is well-formed.

From the polarity of an Infopipe's ports we can construct an expression that represents the polarity of the Infopipe itself. We use a notation reminiscent of a functional type signature. Thus, a buffer, which has a negative Inport and a negative Outport, has a polarity signature $- \rightarrow -$, while a pump, which has two positive ports, has signature $+ \rightarrow +$.

Whereas Buffers seem to be inherently negative and Pumps inherently positive, some components can be modelled equally well with either polarity. For example, consider the function of a defragmenter that combines a pair of information items into a single item. Such functionality could be packaged as a $- \rightarrow +$ Infopipe, which accepts a sequence of two items pushed into its Inport and pushes a single item from its Outport. However, the same functionality could also be packaged as a $+ \rightarrow -$ Infopipe, which `pulls` two items into its Inport and replies to a `pull` request on its Outport with the combined item.

Rather than having two distinct Infopipes with the same functionality but opposite polarities, it is convenient to combine both into a single component, to which we assign the polarity signature $\alpha \rightarrow \bar{\alpha}$. This should be read like a type signature for a polymorphic function, with an implicit universal quantifier introducing the variable α. It means that the ports must have opposite polarities. For example, if a filter with signature $\alpha \rightarrow \bar{\alpha}$ is connected to the Outport of a pump with signature $+ \rightarrow +$, the α would be instantiated as $-$ and the $\bar{\alpha}$ as $+$, and hence the filter would acquire the induced polarity $- \rightarrow +$.

The polarity-checking algorithm that we have implemented is very similar to the usual polymorphic type checking and inference algorithm used for programming languages (Cardelli 1987). The main extension is the addition of a *negation* operation.

3.2 Ensuring Information Flow

Polarity correctness is a necessary condition for information to flow through a pipeline. For example, if two buffers (both with signature $- \rightarrow -$) were directly connected, it would never be possible for information to flow from the first to the second. The polarity check prohibits this. In contrast, a pipeline that contains a pump (with signature $+ \rightarrow +$) between the two buffers will pass the polarity check, and will also permit information to pass from the first buffer to the second.

However, polarity correctness is not by itself sufficient to guarantee timely information flow. In studying these issues, it is useful to think of an Infopipeline as an energy flow system. Initially, energy comes from pumps, and other components with only positive ports, such as positive sources. Eventually, energy will be dissipated in buffers and sinks.

Components such as broadcast tees, which have signature $- \begin{array}{l} \nearrow + \\ \searrow + \end{array}$, can be thought of as amplifying the energy in the information flow, since every information item pushed into the tee causes two items to be pushed out. However, a switching tee, which redirects its input to one or other of its two Outports, is not an amplifier, even though it has the same polarity signature. This can be seen by examining the flows quantitatively. If the input flow has bandwidth b items/s, aggregate output from the broadcast tee is $2b$ items/s, whereas from the switching tee it is b items/s. Similar arguments can be made for droppers and tees that aggregate information; they can be thought of as energy attenuators.

From these considerations we can see that the "energy" flow in a pipeline cannot be ascertained by inspection of the polarities of the components alone. It is also necessary to examine the quantitative properties of the flow through the pipeline, such as information flow rates.

3.3 Buffering, Capacity and Cycles

So far, our discussions have focused on linear pipelines and branching pipelines without cycles. However, we do not wish to eliminate the possibility of cyclic pipelines, where outputs are "recycled" to become inputs. Examples in which cycles may be useful include implementation of chained block ciphers, samplers, and forward error correction.

It appears that a sufficient condition to avoid deadlock and infinite recursion in a cycle is to require that any cycle contains at least one buffer. This condition can easily be ensured by a configuration-checking tool. The polarity check will then also ensure that the cycle contains a pump. However, this rule may not be a necessary condition for all possible implementations of pipeline components, and it remains to be seen if it will disallow pipeline configurations that are useful and would in fact function correctly.

Three other properties that one might like to ensure in a pipeline are (1) that no information items are lost, unless explicitly dropped by a component, (2) that the flow of information does not block, and (3) that no component uses unbounded resources. However, although it may be possible to prove all of these properties for certain flows with known rate and bandwidth, in general it is impossible to maintain all three. This is because a source of unbounded bandwidth can overwhelm whatever Infopipes we assemble to deal with the flow — unless we allow them unbounded resources. We are investigating the use of Queuing Theory models to do quick checks on pipeline capacity.

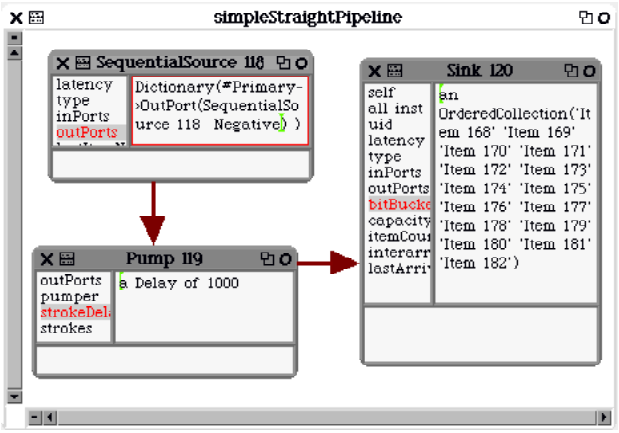


Fig. 6 Inspecting a simple straight pipeline

3.4 The Infopipe Configuration Language

We have prototyped a textual pipeline configuration language by providing Infopipe components with appropriate operators in the Smalltalk implementation. This can be viewed as an implementation of a domain-specific language for pipeline construction by means of a shallow embedding in a host language.

The most important operator for pipeline construction is `->>`, which, as mentioned in section 2.3, is understood by both Infopipes and Ports. This enables simple straight-line infopipes to be built with one line of text, such as `SequentialSource new ->> (p := Pump new) ->> Sink new`. The ability to name the Inports and Outports of an Infopipe explicitly permits us to construct arbitrary topologies with only slightly less convenience, as has already been illustrated in Fig. 4.

Using an existing programming language as a host provides us with a number of benefits, including the use of host language variables to refer to Infopipes, such as `p` in the above example. Because Smalltalk is interactive, the Infopipe programmer can not only start the pipeline (by issuing the control invocation `p startPumping`), but can also debug it using host language facilities. For example, `p inspectPipleine` will open a window (shown in Fig. 6) that allows the programmer to examine and change the state of any of the Infopipes in the pipeline.

4 Implementation Issues

4.1 Threads and Pipes

One of the trickiest issues in implementing Infopipes is the allocation of threads to a pipeline. Port polarity in the Infopipe abstraction has a realtionship to threading, but the relationship is not as simple as it may at first appear.

A component that is implemented with a thread is said to be active. Clearly, a pump is active. In fact, any component that has only positive ports must be active, for there is no

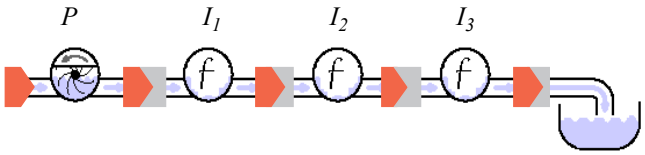


Fig. 7 : A pump drives a series of transformation Infopipes.

other way in which it can acquire a thread to make invocations on other objects.

A very straightforward way of implementing a pump with a frequency f Hz is to generate a new thread every $1/f$ seconds, and to have each such thread execute the code

```
outport push: (inport pull)
```

exactly once. The objection to this approach is that it may generate many threads unnecessarily, and thread creation is often an expensive activity. Moreover, because it is possible for many threads to be active simultaneously, every connected component must behave correctly in the presence of concurrency. In essence, this implementation gives each information item its own thread, and may thus have good cache locality.

An alternative approach is to give the pump a single thread, and to have that thread execute

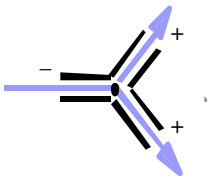
```
outport push: (inport pull).
strokeDelay wait
```

repeatedly, where `strokeDelay wait` suspends the caller for the appropriate inter-stroke interval. However, with the usual synchronous interpretation for method invocation, the pump has no idea how long it will take to execute `outport push:` or `inport pull`. Thus, it cannot know what delay is appropriate: the value of the delay is a property of the pipeline as a whole, not a local property of the pump.

From the perspective of a particular component making a synchronous invocation, the time that elapses between invoking `push:` or `pull` on an adjacent component is an interval in which it has "loaned" its thread to others in the pipeline; we call this interval the *thread latency*. Note that thread latency, like thread, is an implementation-level concept, and is quite distinct from information latency, the time taken for an information item to pass through a component. Thread latency can be reduced by adding additional threads, provided that sufficient CPU time is available, and the CPU scheduler is willing to make it available. Information latency is harder to reduce!

Consider a number of passive components I_1, I_2,... I_n, that are connected in series. Suppose that each I_i has polarity $- \rightarrow +$, and that it performs some transformation on the information that is pushed into it that takes time t_i. The transformed information item is then pushed into component I_{i+1}. If all of the `push` messages are synchronous, the time that elapses between invoking `push:` on I_1 and receiving the reply is given by $t_{\text{total}} = \sum_{i=1}^{n} t_i$.

Now suppose that a pump P is connected to the Inport of I_1, as shown in Fig. 7. If the required interval between strokes of the pump, $t_p = 1/f$, is less than t_{total}, then the single threaded version of the pump will be unable to maintain the specified frequency. It will be necessary to use multiple threads in the pump, and other components in the pipeline will need to incorporate the appropriate synchronization code to deal correctly with this concurrency. (The pump may also need to use multiple CPUs; this depends on the proportion of t_{total} during which the processor is actually busy. If some of the I_n access external devices, t_n may be much greater than the CPU time used by I_n.)



Thread latency is an important parameter not only for pumps but also for other Infopipes. Consider a broadcast *Tee* that accepts an information item at its negative Inport and replicates it at two or more positive Outports. This can be implemented with two threads, which will give the Tee's `push:` method the lowest thread latency, zero threads, in which case the pushes that the Tee performs on its downstream neighbours will be serialized, or one thread, which can be used either to provide concurrency at the Outports, or to reduce the Tee's thread latency at its Inport.

It should now be clear that allocating the right number of threads to a pipeline is not an easy problem. If there are too few, the pipeline may not satisfy its rate specification; if there are too many, we may squander resources in unnecessary bookkeeping and synchronization. Application programmers are relieved of the task of thread allocation by working with Pumps and similar high-level abstractions and dealing instead with application domain concepts such as stroke frequency. But this leaves the Infopipe implementation the responsibility to perform thread allocation.

We have considered two approaches. The first, which we have prototyped, is entirely dynamic. `Pump` uses a timer to wake up after the desired stroke interval. It keeps a stack of spare threads; if a thread is available, it is used to execute the stroke. If no thread is available, a new thread is created. Once the thread has completed the stroke, it adds itself to the stack (or deletes itself if the stack is full).

The second approach, which we have not yet implemented, analyses the pipeline before information starts to flow. The components adjacent to the pump are asked for their thread latencies, the total thread latency for pull and push is computed. If this is less than t_p, we know that a single thread should be sufficient, and simpler single threaded pipeline components can be utilized.

4.2 Creating Polymorphic Infopipes

A Polymorphic Infopipe must have methods for both `pull` and `push:`, and the behaviour of these methods should be coherent, in the sense that the transformation that the Infopipe performs on the information, if any, should be the same in each case.

Although Polymorphic Infopipes are clearly more useful than their monomorphic instances, it is not in general a simple matter to create `push:` and `pull` methods with the required correspondence. Fig. 8 shows sample code for a Defragmenter. We assume that the component has a method `assemble: i1 and: i2` that returns the composite item built from input fragments `i1` and `i2`. The `pull` method, which implements the $+\rightarrow-$ funtionality, and the `push:` method, which implements the $-\rightarrow+$ funtionality, both use

the `assemble:and:` method, but, even so, it is not clear how to verify that `pull` and `push:` both do the same thing.

Indeed, a third implementation style is possible, providing the +→+ polarity; this is shown in Fig. 9. This Defragmenter understands neither `pull` nor `push:`, but instead has an internal thread that repeatedly executes `stroke`.

It is clearly undesirable to have to write multiple forms of the same code, particularly when there must be semantic coherence between them. We can avoid this in various ways.

- Most simply, we can eliminate polymorphic pipes completely. In this situation, the defragmenter would be written with whatever polarity is simplest, probably +→−. If a different polarity is required, this would be constructed as a composition of more primitive infopipes. For example, a buffer, a +→− defragmenter and a pump could be composed to create a −→+ defragmenter.
- The second approach is to use a layer of middleware to "wrap" whichever method is most easily written by hand, in order to generate the other methods. This is possible because the hand-written methods do not send messages to the adjacent components directly, but instead use a level of indirection. For example, the Defragmenter `pull` method sends `pull` to its own `inport` rather than `pull` to the upstream component. A clever implementation of `inport pull` can actually wait for the upstream component to send a `push:` message. We have explored this solution in some depth (Koster et al. 2001b); whenever adjacent Infopipes do not need to be scheduled independently of each other, they are run as coroutines in the same thread, thus avoiding scheduling overhead.
- A third possibility is to automatically transform the source code, so that one version would be written by hand and the others generated automatically. Even with the simple example shown in Fig. 8, this seems to be very hard; in the general case, we do not believe that it is feasible. The difficulty is that the transformation engine would need to "understand" all of the complexity of a general-purpose programming language like Smalltalk or C++.
- It is possible that this objection could be overcome by using a domain-specific source language, with higher-level semantics and more limited expressiveness. From a more abstract form of the method written in such a language, it might be possible to generate executable code in whatever form is required. This approach is currently under investigation.

4.3 Netpipes

Netpipes implement network information flows using whatever mechanisms are appropriate to the underlying medium and the application. For example, we have built a low-latency, unreliable `Netpipe` using UDP.

A `Netpipe` has the same polarity and data interface as a buffer; this models the existence of buffering in the network and in the receiving socket. However, the control interface of a Netpipe is different, since it reflects the properties of the underlying network. For example, the latency of a `Netpipe` depends on the latency of its network connection and the capacity of its buffer.

The motivation for Netpipes to allow Infopipe middleware components in two different address spaces to connect to each other. It is certainly true that an existing distributed computing platform, such as remote method invocation or remote procedure call, would allow such connections. However, if we used such a platform, we could be hiding the communication between the address spaces, and thus giving up any ability to control it — which was the reason that we originally created Infopipes. We would also be violating the seamless connection property described in section 2.1.

However, it is not necessary to reimplement an entire distributed computing environment in order to retain control over the information flow in a Netpipe. Instead, we have bootstraped the Netpipe implementation by using the features of an existing environment, such as naming and remote method invocation.

A `Netpipe` is an Infopipe with an Inport in one address space and an Outport in another, as shown in Fig. 10. This means that the Inport can be in the same address space as its upstream neighbour, and thus invocations of `push:` can use seamless local mechanisms for method invocation. Similarly, the Outport is in the same address space as its downstream

```
Defragmenter >> pull
   | item1 item2 |
   item1 := inport pull.
   item2 := inport pull.
   ↑ self assemble: item1 and: item2.
```

```
Defragmenter >> push: item
   isFirst
      ifTrue:  [buffer := item]
      ifFalse: [
         outport push:
            (self assemble: buffer and: item) ].
```

Fig. 8 : Methods of a polymorphic defragmenter

```
Defragmenter >> stroke
   | item1 item2 |
   item1 := inport pull.
   item2 := inport pull.
   outport put: (self assemble: item1
                           and: item2)
```

```
Defragmenter >> startPumping: period
   self strokeInterval: period.
   [[ self stroke.
      strokeDelay wait] repeatForever] fork
```

Fig. 9 : Defragmenter in the + → + style

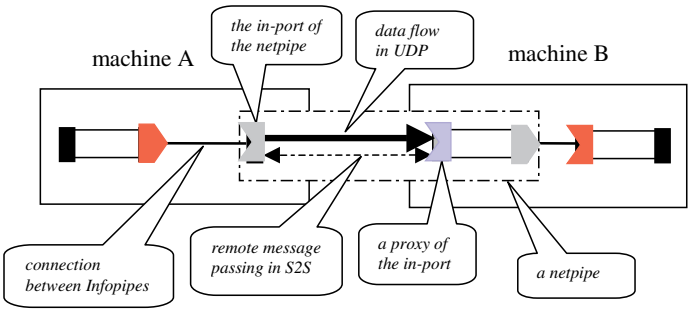


Fig. 10 : Working with a Netpipe

neighbour, which can seamlessly invoke `pull` on the Netpipe. The Netpipe object itself, containing the buffering and the code, is co-located with the Outport.

Our prototyping environment, Squeak Smalltalk (Guzdial 2001; Squeak 2000), is equipped with a remote method invocation package called S2S, which stands for "Squeak to Squeak". S2S provides access transparency and location transparency in a similar way to CORBA and Java RMI. A local proxy can be created for a remote objects; the proxy can then be invoked without callers needing to be aware that they are really using a remote object—except that the call is several orders of magnitude slower.

We take care that we use S2S only to configure and name Infopipe components, and *not* for transmitting information through the pipeline. For example, a `Netpipe` uses S2S to create an Inport on the remote machine, and an S2S proxy for this inport is stored in the Netpipe. However, when the Inport needs to push information into the Netpipe, it uses a custom protocol implemented directly on UDP.

In this way we arrange that Infopipes exhibit *access transparency*: the same protocol is used to establish local and remote Infopipe connections. However, we choose not to provide *location transparency*: connections between adjacent Infopipes must be local, and the `–>>` method checks explicitly that the ports that it is about to connect are co-located. Without this check, ports in different address spaces could be connected directly: information would still flow through the pipeline, but the `push:` or `pull` of each item would require a remote method invocation. As well as being very much less efficient, this would mean that the application would have no control over network communication.

Instead of signalling an error in the face of an attempt to connect non-co-located ports, an alternative solution would be to introduce a Netpipe automatically. We have not pursued this alternative, because in practice it is usually important for the programmer to be aware of the use of the network. For example, it may be necessary to include Infopipe components to monitor the available QoS and adapt the information flow over the Netpipe accordingly.

Fig. 11 shows the code for setting up a MIDI pipeline using a Netpipe. The first two statements obtain S2S proxies for the source and pump objects that already exist on a remote machine called `MusicStore`. We will refer to these proxies as `s` and `p`, The third statement builds a `Netpipe` from `MusicStore` to the local machine. The fifth statement,

```
source := 's2s://MusicStore/source1' asRemoteOjbect.
pump := 's2s://MusicStore/pump1' asRemoteObject.
netPipe := Netpipe from: 's2s://MusicStore/'.
sink := MIDIPlayer new.
source –>> pump –>> netPipe –>> sink.
monitor := Monitor monitored: netPipe
                   controlled: pump.
pump startPumping: 100.
monitor startMonitoring: 1000.
sink startPlaying.
```

Fig. 11 : Code for a streaming MIDI pipeline

`source –>> pump –>>…`, constructs the pipeline. It is interesting to see in detail how this is accomplished.

The invocation `–>>` is sent to `source`, which is a local proxy for remote object `s`. S2S translates this into a remote method invocation on the real object `s` on `MusicStore`. Moreover, because the argument, `pump`, is a proxy for `p`, and `p` is co-located with `s`, S2S will present `p` (rather than a proxy for `pump`) as the argument to the `–>>` invocation. The method for `–>>` will then execute locally to both `s` and `p`, creating a connection with no residual dependencies on the machine that built the pipeline.

A similar thing happens with `netPipe`. Although `netPipe` itself is local, its Inport is on the host `MusicStore`. Thus, the connection between `p` and `netPipe`'s `Inport` is also on `MusicStore`. Information transmitted between `netPipe`'s `Inport` and `OutPort` does of course traverse the network, but it does *not* use S2S; it uses a customized transport that is fully encapsulated in and controlled by `netPipe`.

4.4 Smart Proxies

When information flows from one address space to another, it is necessary not only to agree on the form that the information flow should take, but also to install the Infopipe components necessary to construct that flow. For example, a monitoring application that produces a video stream from a camera should be able to access a logging service that records a video in a file as well as a surveillance service that scans a video stream for suspicious activity. In the case where the video is sent to a file, the file sink and the camera might be on the same machine, and the communication between them might be implemented by a shared-memory pipe. In the case of the surveillance application, the

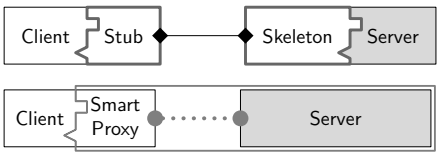


Fig. 12 : Smart Proxies

communication could involve a Netpipe with compression, encryption and feedback mechanisms over the Internet.

Notice that the Infopipe components that need to be co-located with the camera are different in these two cases. Since we wish to allow Infopipes to be dynamically established, we must address the problem of how such components are to be installed and configured.

Koster and Kramp proposed to solve this problem in a client-server environment by using dynamically loadable Smart Proxies (Koster and Kramp 2000). Their idea is that the server functionality is partly implemented on the client node; this enables the server to control the network part of the pipeline, as shown in Fig. 12. At connection setup, the server chooses a communication mechanism based on information about the available resources. A video server, for instance, could use shared memory if it happens to be on the same node as the client, a compression mechanism and UDP across the Internet, or raw Ethernet on a dedicated LAN. It then transmits the code for a Smart Proxy to the address space containing the client. In this way, application specific remote communication can be used without making the network protocol the actual service interface; that would be undesirable because all client applications would have to implement all protocols used by any server to which they may ever connect. Smart Proxies enable the service interface to be described at a high level using an IDL. Client applications can be programmed to this interface as if the server were local, although they actually communicate with the proxy.

The idea of Smart Proxies can be generalized and applied to Infopipes (Koster et al. 2001a). Since there are high-level interfaces between all elements of a pipeline, the granularity of composition can be finer. It is not necessary to send to the consumer address space a monolithic proxy implementing every transformation that needs to be performed on the information stream. Instead, it may be possible to compose the required transformation from standard pipeline elements that may already be available on the consumer side. Thus, it is sufficient to send a description or *blueprint* of the required proxy, and if necessary to send small specialized Infopipes that implement those pieces of the pipeline that are not already available.

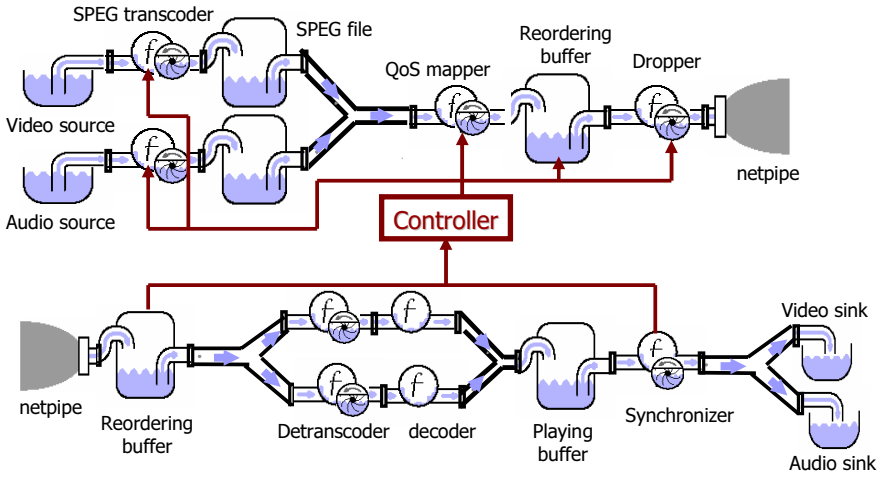
5 Some Example Infopipelines

5.1 The Quasar Video Pipeline

The Quasar video pipeline is a player for adaptive MPEG video streaming over TCP. It supports QoS adaptation in both temporal and spatial dimensions. MPEG-1 video is transcoded into SPEG (Krasic and Walpole 1999) to add spatial scalability through layered quantization of DCT data. To suit the features of TCP, MPEG video is delivered in a *priority-progress stream* (Krasic et al. 2001), which is a sequence of packets, each with a timestamp and a priority. The timestamps expose the timeliness requirements of the stream, and allow progress to be monitored; the priorities allow informed dropping in times of resource overload.

The Quasar video pipeline is shown as an Infopipeline in Fig. 13. At the producer side (the top part of the figure), the video frames first flow through an SPEG transcoding filter, and are buffered. The QoS mapper pulls them from the buffer and gives each packet a priority according to the importance of the packet and the preference of the user. For example, the user might be more concerned with spatial resolution than with frame rate, or *vice versa*. A group of prioritized packets are pushed in priority order into the Reordering buffer. The Dropper is a filter that discards stale packets, and low priority packets, when the network is unable to deliver them in time.

The producer and consumer pipelines are connected by a TCP Netpipe. On the consumer side (at the bottom of the figure) the Reordering buffer arranges packets in time order. The Detranscoder and decoder are filters that convert the packets to MPEG and then to image format, after which they are pushed into the Playing buffer. The Synchronizer pulls them from that buffer at the time that they are required, and presents them to the Video sink. The audio stream is handled



in a similar way; the two streams are merged and split using Tees. The Controller coordinates the rates of all the components through control interfaces.

5.2 The MIDI Pipeline

The MIDI pipeline (see Fig. 14) was built using some existing libraries from the Squeak Smalltalk system. The Squeak MIDI player buffered an entire MIDI file before playing it. We adapted this player to deal with streaming data and wrapped it as three Infopipe components: the MIDISource, the MIDIFilter, and the MIDISink.

The MIDISource reads "note-on" and "note-off" commands from a MIDI file; the MIDIFilter combines a note-on command and its corresponding note-off command to generate a note event, which consists of a key and its duration. The MIDISink plays a stream of note events. To make the MIDI player stream over the Internet, we needed only to insert two pre-existing Infopipe components: a pump and a UDP netpipe. To ensure that the MIDISink plays smoothly, we added a controller that monitors the fill level of the Netpipe and adjusts the pumping rate accordingly. In this prototype, the connections to the controller were not implemented using Infopipes; whether they should be is an open question. Instead, we used direct method invocation, which means that invocations from the controller to the pump used S2S, which is very much slower than an Infopipe. We found that a buffer sufficient to hold 30 note events produced smooth playout while still minimizing the number of control messages.

6 Related Work

Some related work aims at integrating streaming services with middleware platforms based on remote method invocations such as CORBA. The CORBA Telecoms specification (OMG 1998a) defines stream management interfaces, but not the data transmission. Only extensions to CORBA such as TAO's pluggable protocol framework allow the use of different transport protocols and, hence, the efficient implementation of audio and video applications (Mungee et al. 1999). Asynchronous messaging (OMG 2001a) and event channels (OMG 2001b) allow evading the synchronous RMI-based interaction and introduce the concurrency needed in an information pipeline. Finally, Real-time CORBA (OMG 2001a; Schmidt and Kuhns 2000), adds priority-based mechanisms to support predictable service quality end to end. As extensions of an RMI-based architecture these mechanisms facilitate the integration of streams into a distributed object system. Infopipes, however, provide a high-level interface tailored to information flows and more flexibility in controlling concurrency and pipeline setup.

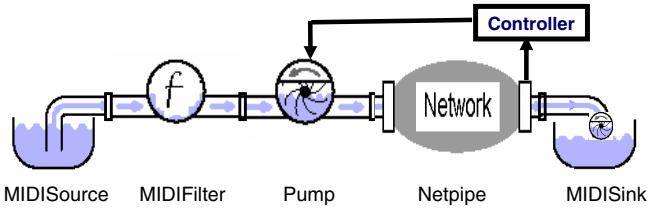


Fig. 14 : The MIDI pipeline

Structuring data processing applications as components that run asynchronously and communicate by passing on streams of data items is a common pattern in concurrent programming (see, for example, reference (Lea 1997)). Flow-Based Programming applies this concept to the development of business applications (Morrison 1994). While the flow-based structure is well-suited for building multimedia applications, it must be supplemented by support for timing requirements. Besides integrating this timing control via pumps and buffers, Infopipes facilitate component development and pipeline setup by providing a framework for communication and threading.

QoSDREAM uses a two-layer representation to construct multimedia applications (Naguib and Coulouris 2001). On the model layer, the programmer builds the application by combining abstract components and specifying their QoS properties. The system then maps this description to the active layer consisting of the actual executable components. The setup procedure includes integrity checks and admission tests. The active-layer representation may be more fine-grained than the model specification introducing additional components such as filters, if needed. In this way, the system supports partially automatic configuration. While the current Infopipe implementation provides less sophisticated QoS control, it provides a better modelling of flow properties by explicitly using pumps and buffers.

Blair and co-workers have proposed an open architecture for next-generation middleware (Blair et al. 1998; Eliassen et al. 1999). They present an elegant way to support open engineering and adaptation using reflection, a technique borrowed from the field of programming languages (Blair and Coulson 1998). In their multimedia middleware system, TOAST (Eliassen et al. 2000; Fitzpatrick et al. 2001) they reify communication through open bindings, which are similar to our remote pipes. The scope of this work is wider than that of Infopipes, which are specialized for streaming applications.

The MULTE middleware project also features open bindings (Eliassen et al. 2000; Plagemann et al. 2000) and supports flexible QoS (Kristensen and Plagemann 2000). It provides applications with several ways to specify QoS using a mapping or negotiation in advance to translate among different levels of QoS specification. In our approach we typically use dynamic monitoring and adaptation of QoS at the application-level to implicitly manage resource-level QoS.

Ensemble (van Renesse et al. 1997) and Da CaPo (Vogt et al. 1993) are protocol frameworks that support the composition and reconfiguration of protocol stacks from modules. Both provide mechanisms to check the usability of configurations and automatically configure the stacks. Unlike these frameworks for local protocols, Infopipes use a uniform abstraction for handling information flows from source to sink, possibly across several network nodes; the Infopipe setup is controlled by the application. A similarity is that both allow for dynamic configuration: protocol frameworks dynamically (re-)configure protocol stacks between a network interface and an application interface, while Smart Proxies dynamically construct part of an Infopipeline between a client-side service interface and a remote server, providing protocol-independent service access.

The Scout operating system (Mosberger and Peterson 1996) combines linear flows of data into paths. Paths provide an abstraction to which the invariants associated with the flow can be attached. These invariants represent information that is true of the path as a whole, but which may not be apparent to any particular component acting only on local information. This idea—providing an abstraction that can be used to transmit non-local information—is applicable to many aspects of information flows, and is one of the principles that Infopipes seek to exploit. For instance, in Scout paths are the unit of scheduling, and a path, representing all of the processing steps along its length, makes information about all of those steps available to the scheduler.

7 Summary and Future Work

Infopipes are a subject of continuing research; the work described here does not pretend to be complete, although early results have been encouraging. The applications that have driven the work described here have primarily been streaming video and audio. However, Infopipes also form part of the communications infrastructure of the Infosphere project (Liu et al. 2000; Pu et al. 2001), and we intend that Infopipes are also useful for applications such as environmental observation and forecasting (Steere et al. 2000) and continual queries (Liu et al. 1999).

We have been pursuing three threads of research simultaneously. The first, which pre-dates the development of Infopipes themselves, is the design and implementation of a series of video players that stream video over the Internet, adapting their behaviour to make the best possible use of the available bandwidth (Cen et al. 1995; Cowan et al. 1995; Inouye et al. 1997; Koster 1996; Krasic and Walpole 2001; Staehli et al. 1995). The second thread is related to the underlying technologies that support streaming media, in particular, adaptive and rate-sensitive resource scheduling (Li et al. 2000; Steere et al. 1999a; Steere et al. 1999b) and congestion control (Cen et al. 1998; Li et al. 2001a; Li et al. 2001b). It is these technologies that enable us to design and build the Infopipes that are necessary for interesting applications.

The final thread is a prototyping effort that has explored possible interfaces for Infopipes in an object-oriented setting. We have used Squeak Smalltalk as a research vehicle; this has been a very productive choice, as it enabled us to quickly try out — and discard — many alternative interfaces for Infopipes before settling on those described here. The Squeak implementation is not real-time, but it is quite adequate for the streaming MIDI application (section 5.2).

We are currently embarked on the next stage of this research, which involves weaving these threads together into a fabric that will provide a new set of abstractions for streaming applications. We are in the process of using the Infopipe abstractions described here to re-implement our video pipelines on a range of platforms including desktop, laptop and wireless handheld computers as well as a mobile robot. We are also exploring kernel-level support for Infopipes under Linux, with a view to providing more precise timing control and an application-friendly interface for timing-sensitive communication and device I/O.

Acknowledgements. This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs, by NSF Grant CCR-9988440, by the Murdock Trust, and by Intel. We thank Paul McKenney for useful discussions and Nathanael Schärli for help with the Squeak graphics code.

References

Blair GS, Coulson G (1998) The case for reflective middleware. Internal report MPG-98-38, Distributed Multimedia Research Group, Department of Computing, Lancaster University, Lancaster, UK

Blair GS, Coulson G, Robin P, Papathomas M (1998) An Architecture for Next Generation Middleware. In: Davies N, Raymond K, Seitz J, eds. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware' 98), Lake District, UK. Springer Verlag

Cardelli L (1987). Basic Polymorphic Typechecking. Science of Computer Programming 8(2)

Cen S, Pu C, Staehli R, Cowan C, Walpole J (1995) A Distributed Real-Time MPEG Video Audio Player. Fifth International Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV'95), Durham, New Hampshire, USA. Lecture Notes in Computer Science Vol. 1018. Springer Verlag

Cen S, Pu C, Walpole J (1998) Flow and Congestion Control for Internet Streaming Applications. Proceedings Multimedia Computing and Networking (MMCN98)

Chesire M, Wolman A, Voelker GM, Levy HM (2001) Measurement and Analysis of a Streaming Media Workload. USENIX Symposium on Internet Technologies and Systems (USITS), San Francisco, CA, USA

Cowan C, Cen S, Walpole J, Pu C (1995). Adaptive Methods for Distributed Video Presentation. ACM Computing Surveys 27(4):580-583

Eliassen F, Andersen A, Blair GS, et al. (1999) Next Generation Middleware: Requirements, Architecture, and Prototypes. 7th Workshop on Future Trends of Distributed Computing Systems (FTDCS'99), Cape Town, South-Africa

Eliassen F, Kristensen T, Plagemann T, Raffaelsen HO (2000) MULTE-ORB: Adaptive QoS Aware Binding. International Workshop on Reflective Middleware (RM 2000), New York, USA

Fitzpatrick T, Gallop J, Blair G, Cooper C, Coulson G, Duce D, Johnson I (2001) Design and Application of TOAST: An Adaptive Distributed Multimedia Middleware Platform. Interactive Distributed Multimedia Systems (IDMS 2001), Lancaster, UK. Lecture Notes in Computer Science Vol. 2158. Springer Verlag

Guzdial M (2001) Squeak: Object-oriented Design with Multimedia Applications. Upper Saddle River, NJ: Prentice Hall

Inouye J, Cen S, Pu C, Walpole J (1997) System Support for Mobile Multimedia Applications. 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97), St. Louis, Missouri

ISO (1998) Information Technology: Open Distributed Processing. ISO Standard ISO/IEC 10746, International Standards Organization

Jacobs S, Eleftheriadis A (1998). Streaming video using dynamic rate shaping and TCP flow control. Journal of Visual Communication and Image Representation

Karr DA, Rodrigues C, Loyall JP, Schantz RE, Krishnamurthy Y, Pyarali I, Schmidt DC (2001) Application of the QuO Quality-of-Service Framework to a Distributed Video Application. International Symposium on Distributed Objects and Applications, Rome, Italy

Koster R.(1996) Design of a Multmedia Player with Advanced QoS Control [M.S. Thesis]. Oregon Graduate Institute of Science & Technology, Beaverton, OR, USA

Koster R, Black AP, Huang J, Walpole J, Pu C (2001a) Infopipes for Composing Distributed Information Flows. International Workshop on Multimedia Middleware. ACM Press

Koster R, Black AP, Huang J, Walpole J, Pu C (2001b) Thread Transparency in Information Flow Middleware. In: Guerraoui R, ed. Middleware 2001—IFIP/ACM International Conference on Distributed Systems Platforms, Heidelberg, Germany. Lecture Notes in Computer Science Vol. 2218. Springer Verlag

Koster R, Kramp T (2000) Structuring QoS-supporting services with Smart Proxies. Second International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000). Lecture Notes in Computer Science Vol. 1795. Springer Verlag

Krasic B, Walpole J (2001) Priority-Progress Streaming for Quality-Adaptive Multimedia. ACM Multimedia Doctoral Symposium, Ottawa, Canada

Krasic C, Li K, Walpole J (2001) The Case for Streaming Multimedia with TCP. 8th International Workshop on Interactive Distributed Multimedia Systems — iDMS 2001, Lancaster, UK. Lecture Notes in Computer Science Vol. 2158. Springer Verlag

Krasic C, Walpole J (1999) QoS Scalability for Streamed Media Delivery. Technical Report CSE-99-11, Department of Computer Science & Engineering, Oregon Graduate Institute, Beaverton, OR

Kristensen T, Plagemann T (2000) Enabling Flexible QoS Support in the Object Request Broker COOL. IEEE ICDCS International Workshop on Distributed Real-Time Systems (IWDRS 2000), Taipei, Taiwan, Republic of China

Lea D (1997) Concurrent Programming in Java. Addison-Wesley

Li K, Krasic C, Walpole J, Shor M, Pu C (2001a) The Minimal Buffering Requirements of Congestion Controlled Interactive Multimedia Applications. 8th International Workshop on Interactive Distributed Multimedia Systems — iDMS 2001, Lancaster, UK. Lecture Notes in Computer Science Vol. 2158. Springer Verlag

Li K, Shor M, Walpole J, Pu C, Steere D (2001b) Modeling the Effect of Short-term Rate Variations on TCP-Friendly Congestion Control Behavior. American Control Conference, Alexandria, Virginia

Li K, Walpole J, McNamee D, Pu C, Steere DC (2000) A Rate-Matching Packet Scheduler for Real-Rate Applications. Multimedia Computing and Networking Conference (MMCN'2000), San Jose, California

Liu L, Pu C, Schwan K, Walpole J (2000). InfoFilter: Supporting Quality of Service for Fresh Information Delivery. New Generation Computing Journal 18(4)

Liu L, Pu C, Tang W (1999). Continual Queries for Internet Scale Event-Driven Information Delivery. IEEE Transactions on Knowledge and Data Engineering 11(4)

McCanne S, Vetterli M, Jacobson V (1997). Low-complexity video coding for receiver-driven layered multicast. IEEE Journal on Selected Areas in Communications 16(6):983–1001

Morrison JP (1994) Flow-Based Programming : A New Approach to Application Development. Van Nostrand Reinhold

Mosberger D, Peterson LL (1996) Making paths explicit in the Scout operating system. Second USENIX symposium on Operating systems design and implementation (OSDI).

Mungee S, Surendran N, Krishnamurthy Y, Schmidt DC (1999) The Design and Performance of a CORBA Audio/Video Streaming Service. Hawaiian International Conference on System Sciences (HICSS), Hawaii

Naguib H, Coulouris G (2001) Towards automatically configurable multimedia applications. International Workshop on Multimedia Middleware, Ottawa, Canada

OMG (1998a) CORBA telecoms specification. Object Management Group, Framingham, MA, USA. http://www.omg.org/cgi-bin/doc?formal/98-07-12

OMG (1998b) CORBA/IIOP 2.3 Specification. OMG Document formal/98-12-01, Object Management Group, Framingham, MA, USA. http://www.omg.org/

OMG (2001a) The Common Object Request Broker: Architecture and Specification. Object Management Group, Framingham, MA, USA. http://www.omg.org/cgi-bin/doc?formal/01-09-34

OMG (2001b) Event service specification. Object Management Group, Framingham, MA, USA. http://www.omg.org/cgi-bin/doc?formal/01-03-01

OSF (1991) Remote Procedure Call in a Distributed Computing Environment: A White Paper. Open Software Foundation

Plagemann T, Eliassen F, Hafskjold B, Kristensen T, Macdonald RH, Rafaelsen HO (2000) Managing Cross-Cutting QoS Issues in MULTE Middleware. ECOOP Workshop on Quality of Service in Distributed Object Systems, Sophia Antipolis and Cannes, France

Pu C, Schwan K, Walpole J (2001). Infosphere Project: System Support for Information Flow Applications. ACM SIGMOD Record 30(1)

Schmidt DC, Kuhns F (2000). An overview of the real-time CORBA specification. IEEE Computer 33(6):56-63

Squeak (2000) Squeak. home page, http://www.squeak.org/ Squeak Foundation

Staehli R, Walpole J, Maier D (1995). Quality of Service Specification for Multimedia Presentations. Multimedia Systems 3(5/6)

Steere D, Baptista A, McNamee D, Pu C, Walpole J (2000) Research Challenges in Environmental Observation and Forecasting Systems. Mobicom 2000.

Steere DC, Goel A, Gruenberg J, McNamee D, Pu C, Walpole J (1999a) A Feedback-driven Proportion Allocator for Real-Rate Scheduling. Operating System Design and Implementation (OSDI' 99)

Steere DC, Walpole J, Pu C (1999b) Automating Proportion/Period Scheduling. 20th IEEE Real-Time Systems Symposium, Phoenix, Arizona, USA

Sun (2002) Java Remote Method Invocation Specification. Java™ 2 SDK v1.4, Standard Edition. Web document, http://java.sun.com/j2se/1.4/docs/guide/rmi/spec/rmiTOC.html Sun Microsystems Corp.

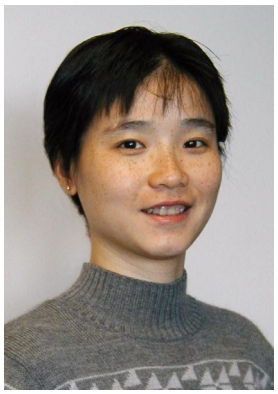
Thompson K, Miller GJ, Wilder R (1997). Wide-area internet traffic patterns and characteristics. IEEE Network Magazine 11(6):10–23

van Renesse R, Birman K, Hayden M, Vaysburd A, Karr D (1997) Building adaptive systems using Ensemble. Technical Report TR97-1638, Computer Science Department, Cornell University

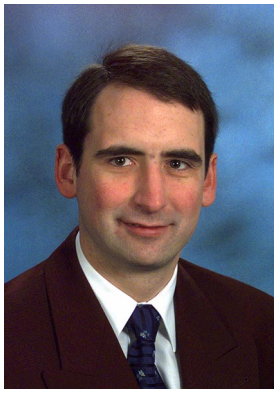
Vogt M, Plattner B, Plagemann T, Walter T (1993) A Run-time Environment for Da CaPo. INET '93. Internet Society

Walpole J, Koster R, Cen S, Cowan C, Maier D, McNamee D, Pu C, Steere D, Yu L (1997) A Player for Adaptive MPEG Video Streaming over the Internet. 26th Applied Imagery Pattern Recognition Workshop AIPR-97, Washington, DC. SPIE

ANDREW P. BLACK holds a D.Phil in Computation from the University of Oxford. At the University of Washington (1981-1986) he was part of a team that built two of the earliest distributed object-oriented systems. From 1987 until 1994 he was with the Distributed Systems Advanced Development group and the Cambridge Research Laboratory of Digital Equipment Corporation. Subsequently, he joined the faculty of the Oregon Graduate Institute as Professor and Head of the Computer Science Department. Since 2000 he has been pursuing his research interests in programming languages, programming methodology, and system support for distributed computing.



JIE HUANG received a B.S. degree in Computer and Communications in 1992 and an M.S. degree in Computer Science in 1995, both from Beijing University of Posts and Telecommunications. She then held the post of Assistant Professor at the same school. Since September 1999 she has been a Ph.D. student at the Oregon Graduate Institute, now the Oregon Health & Science University. Her interests are in software development methodology and programming languages, especially a domain-specific approach for building multimedia networking applications.



RAINER KOSTER received a Master of Science in Computer Science and Engineering in 1997 from the Oregon Graduate Institute of Science and Technology and a Diplom in Computer Science in 1998 from the University of Kaiserslautern. He currently is a member of the Distributed Systems Group at the University of Kaiserslautern. His interests and research focus on quality-of-service support and distributed multimedia systems.

JONATHAN WALPOLE received his Ph.D. in Computer Science from Lancaster University, UK, in 1987. He worked for two years as a post-doctoral research fellow at Lancaster University before taking a faculty position at the Oregon Graduate Institute (OGI). He is now a Full Professor and Director of the Systems Software Laboratory at the OGI School of Science and Engineering at Oregon Health & Science University. His research interests are in operating systems, distributed systems, multimedia computing, and environmental information technology.

CALTON PU received his Ph.D. in Computer Science from University of Washington in 1986, and has served on the faculty of Columbia University and the Oregon Graduate Institute. He is currently a Professor at the College of Computing, Georgia Institute of Technology where he occupies the John P. Imlay, Jr. Chair in Software, and is a co-director of the Center for Experimental Research in Computer Systems. Dr. Pu leads the Infosphere project, a collaboration between Georgia Tech. and OGI that is building systems support for information-driven distributed applications; his other research interests include operating systems, transaction processing and Internet data management.

APPENDIX J

Adaptive Live Video Streaming by Priority Drop. Jie Huang, Charles Krasic, Jonathan Walpole, and Wuchi Feng, IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS 2003), Miami, FL, July 2003.

Adaptive Live Video Streaming by Priority Drop

Jie Huang, Charles Krasic, Jonathan Walpole, and Wu-chi Feng
OGI School of Science and Engineering
Oregon Health and Science University
{jiehuang, krasic, walpole, wuchi}@cse.ogi.edu

Abstract

In this paper we explore the use of Priority-progress streaming (PPS) for video surveillance applications. PPS is an adaptive streaming technique for the delivery of continuous media over variable bit-rate channels. It is based on the simple idea of reordering media components within a time window into priority order before transmission. The main concern when using PPS for live video streaming is the time delay introduced by reordering. In this paper we describe how PPS can be extended to support live streaming and show that the delay inherent in the approach can be tuned to satisfy a wide range of latency constraints while supporting fine-grain adaptation.

1. Introduction

Scalable video surveillance systems, where potentially thousands of cameras are involved, will require the underlying networking and coding mechanism to be scalable, efficient, and adaptive. In particular, the video cameras in these systems in aggregate can easily overload the network that connects them. As a result, we expect that in such systems small computing resources will be placed with each camera that allows it to deal with the resource constraints. We believe that such resources should be used to help make the system as scalable as possible and to provide the highest quality video.

Priority-progress streaming (PPS) is such an adaptive streaming mechanism [5] [6]. It uses a time-window-based approach in which all data packets with timestamps within a certain period of time are placed in a window and reordered into priority order before transmission. It then transmits these packets for the time duration of the window only. At the end of the window duration, it discards unsent packets and moves on to the next window. In this way, the available bandwidth is used to send the most important elements of the stream and the least important elements are dropped. Our implementation of PPS in the Quasar video pipeline [5] shows that PPS is good for streaming stored video.

The reordering window in PPS introduces latency, however, and this latency might be problematic for video surveillance applications, which stream live video. In live video streaming, the latency characteristics of the streaming mechanisms partially determine the freshness of the video content. The freshness of the video content, which is measured by the end-to-end latency from a frame being captured to its display, tends to be important for video surveillance applications.

In this paper, we explore how much of a problem the latency in PPS is for live video streaming and determine the range of video surveillance applications it can support. We describe how PPS can be extended to support live video streaming, and evaluate the latency implications of the approach. Our implementation of the live Quasar pipeline shows that even with fairly rudimentary scalable video encoding technology, the latency due to adaptation in PPS can be reduced to as little as 400ms while maintaining fine-grain adaptation. This means that applications with a latency tolerance of a half second can be supported using TCP-friendly protocols on a coast to coast link in the US (where propagation delay is typically less than 100ms).

This paper is organized as follows. Related work is discussed in Section 2. Section 3 introduces the basic idea of PPS and describes how it works for stored video streaming. Section 4 discusses the problems of using PPS for live video streaming. Section 5 outlines a series of experiments for evaluating the latency and adaptation granularity characteristics of PPS and presents results. Finally, Section 6 concludes the paper and discusses future work.

2. Related work

Streaming video adaptively involves many research areas. Wu et al. have written a comprehensive survey of video streaming approaches and directions in their paper [11]. Vandalore et al. give a detailed survey of application level adaptation techniques [10].

A common approach to adaptive live streaming is to monitor network conditions using feedback-based

mechanisms such as RTCP receiver reports [1] and adjust video encoding parameters on the fly [4][8] so that the rate of the encoded video stream matches a dynamically determined target bandwidth. A key advantage of this approach is compression efficiency—the video encoder is able to optimize video quality for the given target bandwidth. Another advantage is its support for fine-grain adaptation—the target bandwidth can be chosen from a continuous range. A third advantage is low latency—adaptation can be performed without reordering data. The main disadvantages of the approach are its inability to satisfy conflicting requirements of heterogeneous receivers in a simulcast or multicast distribution network, the difficulty of tuning encoding parameters to achieve optimal video quality for a certain video rate, and the difficulty of tuning the feedback control to determine the suitable and accurate target video rate. If the target video rate is chosen incorrectly it will either result in network underutilization, congestion, or increased delay.

PPS takes an alternative approach based on scalable video encoding and priority dropping. Without dynamically manipulating encoding parameters, a scalable encoding approach allows a wide range of video rates at the expense of some compression efficiency. Adaptation is supported in PPS by prioritizing data in the scalable video stream and dynamically dropping data in priority order in order to match the target bandwidth. PPS's sending strategy does not rely on complex control models and is independent of receiver feedback. Instead, it allows an underlying congestion control protocol, such as TCP or any of the TCP-friendly streaming protocols [9] to determine the appropriate sending rate. Whatever that rate is, PPS sends video packets in priority order from a window as fast as possible. In this way, a high-bandwidth receiver gets more data than a low-bandwidth receiver for each window. They both get the best possible video quality under their bandwidth limitations because for either receiver, the data packets received are more useful than those discarded, and the maximum possible bandwidth is used while preserving TCP-friendly behavior. A key advantage of this approach is the simplicity of the mechanisms and the ability to support heterogeneous simulcast distribution efficiently.

3. Priority-progress streaming

3.1. Basic streaming

PPS uses timestamps and priority labels to perform adaptive streaming. A window in PPS contains all data packets with timestamps within a certain period of time. The window is called an adaptation window; and the adaptation window size is the time duration, not the number of packets or number of bytes in this window.

Figure 1 shows an ideal example of PPS streaming with sufficient bandwidth and a constant delay. Data packets with timestamps and priority labels are grouped into windows in time order. Suppose the timestamps are in milliseconds, and the window size is 100ms. Within each 100-ms window, packets are sorted and sent in priority order, assuming that a small number represents a high priority. Packets in a window are sent out as fast as possible. Hence, when PPS runs over TCP, it can deal with TCP's burstiness. In this example, the bandwidth is higher than the data rate, so there is spare time in the 100-ms window. The spare time can be used for work-ahead or bandwidth skimming [5].

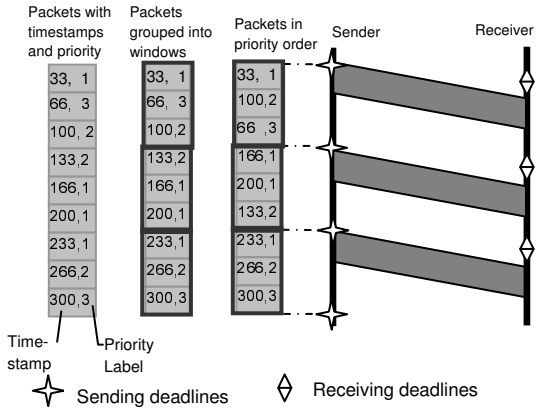


Figure 1. PPS streaming

3.2. Adaptation

As shown in Figure 2, PPS can adapt to the available bandwidth. If the bandwidth is lower than the data rate, some data packets are unsent when the window time expires. These data packets, which have low priorities, are dropped. PPS makes efficient use of the limited bandwidth by transferring the data packets with highest priority first.

Figure 3 (a) shows how PPS deals with increased delay by asking the sender to send data earlier so that it has more time to reach the receiver. If the delay decreases, PPS could either change back to the old sending schedule to keep the receiver buffer fill-level low, or keep the current schedule so as to prime the receiver-side buffer in anticipation of future delay and bandwidth variations.

In practice, the two adaptation mechanisms cooperate to match the varying network conditions.

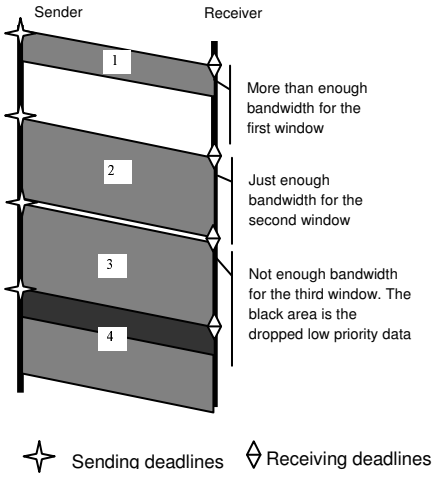


Figure 2. Adaptation to bandwidth

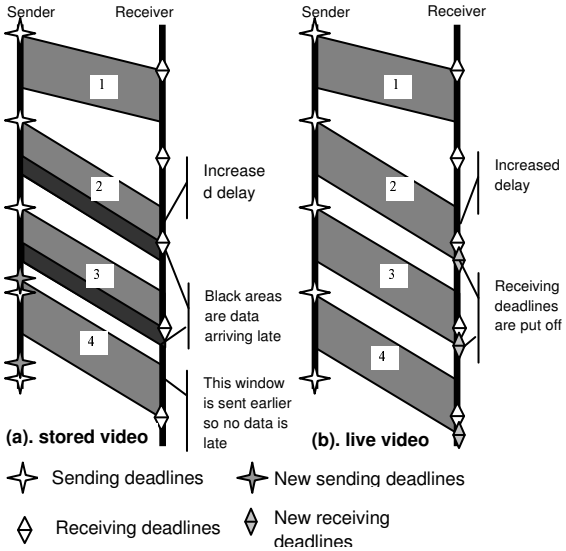


Figure 3. Adaptation to delay

3.3. Preparation for video streaming

PPS can be used to stream any data flow that can be packetized such that each packet can be time-stamped and prioritized. In this section, we discuss packetization, timestamping, and prioritization for video streams.

A video stream consists of video frames; the video frames could be the data packets for PPS. However, how the video frames are encoded determines the space for adaptation. Scalable encoding is preferred because the video stream can work at different data rates and we can achieve different quality levels under different network conditions. The Quasar pipeline uses a scalable compression format called SPEG (Scalable MPEG), extending MPEG-1 video with SNR scalability [5]. Each MPEG video frame is divided into four layers, in which the base layer contains the most significant bits of the DCT coefficients and the successive layers contain the less significant bits. Each layer of an MPEG frame is encapsulated in an SPEG packet.

Video frames have inherent timestamps: the play time. SPEG packets are given the timestamps of the corresponding MPEG frames.

Prioritization enables PPS to do wise adaptation without understanding the complex semantics of video encoding. In the Quasar pipeline, prioritization exposes temporal scalability and SNR scalability by reflecting dependencies among SPEG packets. For example, the base layer of an I frame has higher priority than the base layers of any P frames that depend on it, and a base layer has higher priority than the enhancement layers in the same MPEG frame. However, dependencies decide only a partial order among SPEG packets. For the importance of the base layer of a P frame over an enhancement layer of an I frame, Quasar's prioritization mechanism takes into account how much a user prefers frame rate over picture SNR. This mechanism does not simply assign a priority according to the frame type and layer; instead, it uses a window-based scheme resulting in many more priority levels for a video stream than a one-frame-based algorithm and hence supports much finer-grain adaptation. The larger the window, the more priority levels can be utilized. We combine some quality levels that are indistinguishable by human eyes and we define at most 16 priority levels at any given time in the Quasar pipeline. Details of the algorithm can be found in our earlier papers [5].

SPEG is just an example scalable video format. Subsequently new scalable video encoding approaches, such as MPEG-4 FGS, have better compression efficiency and finer-grain scalability than SPEG and hence offer an even more favorable platform for PPS.

4. Live streaming

4.1. Adaptation for live video

Using PPS for live video introduces much more than simply replacing the stored video file with a video camera. A big difference between stored video and live video is that live video has its own capture clock. Hence, live video cannot be generated faster or slower than its capture rate, while stored video can be read whenever it is needed. This difference implies that the work-ahead mechanism described in Section 3.2 for dealing with increased delay cannot be used for live video. For live video, since it is not possible to "read ahead", it cannot be sent ahead; we instead introduce delay at the receiver by pushing back the receiving deadlines, as shown in Figure 3 (b).

Note that this mechanism is suitable for multicast because each receiver can adjust its receiving deadlines to compensate for its own network delay.

4.2. Latency for streaming

The capture clock introduces the notion of end-to-end latency, which is the time from a frame being captured to its being displayed. Reducing this end-to-end latency is a goal specific to live video streaming. For stored video streaming, applications do require that frames arrive on time for display, but it does not matter when the frame is read from a file or how long it stays in buffers as long as it is on time for display.

There are two main sources of latency: the end machines and the network. Latency from the end machines includes the processing time and the buffering time. On both the sender and the receiver, the processing time does not vary much. For example, the time for encoding, decoding, reordering, and prioritizing is fixed unless we improve the algorithms or switch to faster computers. Therefore we can assume that in general these times are fixed.

Two types of buffers contribute to the total buffering time. Some buffers enable asynchrony among pipeline components. For example, the capture buffer keeps raw video frames from being dropped while the CPU is occupied by encoding or prioritizing; similarly the display buffer allows a smooth playback when a complex frame takes longer than its display duration to decode. These buffers need only be large enough to prevent the pipeline from stalling. The other type of buffer permits adaptation. The time spent in these buffers is determined by the adaptation window size, which is an adjustable PPS parameter.

The latency of the network segment is something that we adapt to and cannot control.

Ignoring the latency sources that are independent of PPS, the end-to-end latency due to adaptation buffers is the sum of the adaptation window size and the transmission time, as shown in Figure 4. For PPS adaptation, the last packet in the window is delayed on the sender side for the whole window time but not delayed at all on the receiver side. Similarly, the first packet in the window is delayed on the receiver side but not at all on the sender side. All of the frames in between are delayed both on the sender and the receiver, but the total delay is always one window time. The transmission time is related to the window size in two ways. When the bandwidth is higher than the data rate, the transmission time is proportional to the amount of data in a window, which is proportional to the window size. When the bandwidth is lower than the data rate, the window size is the transmission time for this window, according to the PPS streaming algorithm (since transmission continues for the entire duration of the window before data is dropped).

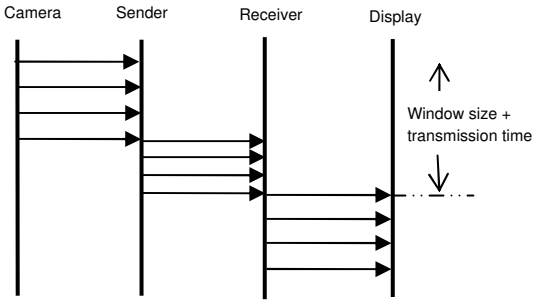


Figure 4. End-to-end latency

In summary, for the normal case when bandwidth is limited, the latency inherent in PPS is generally twice the window size. Thus, tuning the adaptation window size is the key to tuning the end-to-end latency. For low latency streaming, a small window size is preferable. However, a small window size makes fine-grain adaptation difficult and eventually impacts video quality. This is because adaptation happens within a window, i.e. a smaller window provides fewer droppable data units and fewer priority levels for adaptation.

5. Experiments

We have implemented the live Quasar pipeline by extending PPS for live streaming and substituting the MPEG source and the SPEG transcoding components of the pipeline with a camera, a capture card, and a software SPEG encoder. The capture card we use is a WinTV card from Hauppauge. The SPEG encoder is based on ffmpeg [2], an open source encoder that can encode in real time. We modified ffmpeg to implement SPEG's SNR layering strategy and to produce SPEG output directly to the live Quasar pipeline.

The live Quasar pipeline runs on Linux Mandrake 8.1. The sender and the receiver are two Pentium III 930MHz machines. The transport protocol we use is TCP. We run the pipeline on a private LAN without any competing traffic. We also maintain minimum buffer fill levels that allow the pipeline to run smoothly. Thus the adaptation window size is the main control variable in the experiments.

Measurements are obtained through the gscope software oscilloscope [3], which is a time-sensitive visualization tool that shows the bandwidth usage, buffer fill level, end-to-end latency, and other signals in real time.

In the following subsections, we concentrate on the relationships between end-to-end latency, the adaptation window size, and the adaptation granularity. We use the adaptation granularity as an indication of the effectiveness

of the adaptation. The adaptation granularity determines how closely a pipeline can utilize a given level of resource capacity, which is bandwidth in our experiments.

5.1. Latency vs. window size

Figure 5 shows the relationship between the latency and the adaptation window size. As expected, the latency grows with the window size. From Figure 5 we can see that when the adaptation window size is less than 167ms, the latency from adaptation, plus processing and necessary buffering, is well below 400ms. In the real world, the end-to-end latency also includes the network propagation delay and transmission time.

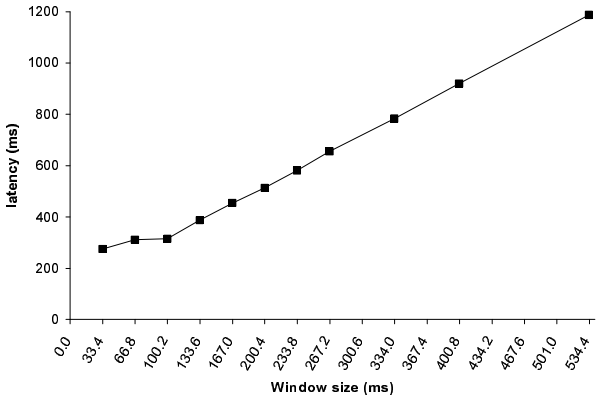


Figure 5. Latency vs. adaptation window size

The latencies shown in Figure 5 are measured for an intra-encoded video stream. The latencies for inter-encoded streams are very close to those shown in Figure 5 and the GOP size has little impact on the end-to-end latency. The window size is the determinant factor.

5.2. Adaptation granularity vs. window size

Each window size can deliver a certain number of possible quality levels. These quality levels range from full quality, when all packets of the window are delivered, to zero quality when none is delivered. Between these two extremes lie a number of quality levels, one for each priority, whose bandwidth requirements can be represented as a percentage of the full quality video bandwidth. As discussed in Section 2.3, the number of priority levels and their corresponding bandwidth percentages depend on the scalability of the video stream, the window size, and the user preferences.

In Figure 6, Figure 7, and Figure 8, we show samples of quality levels available for different window sizes and user preferences. Each symbol + in the plot area represents a quality level. The x value of the symbol is the window size in which that quality level is available; the y value of the symbol is the percentage of the full quality video bandwidth for that quality level. Figure 6 shows the available quality levels when a user prefers temporal quality and SNR quality equally; Figure 7 shows the available quality levels when a user prefers the maximum temporal quality; and Figure 8 shows the available quality levels when a user prefers the maximum SNR quality.

Ideally, for each window size there should be many available quality levels and their bandwidth percentages should be evenly distributed in order to closely match the varying network bandwidth. However, the scalability of video encoding and the window size determine how many prioritizable and independently droppable units are in a window and the sizes of these units determine the distribution of bandwidth percentage for quality levels. For the window size of 33.4ms, each window includes only one MPEG frame at NTSC rate. If no scalability is introduced, there is only one droppable unit in the window and there is only one quality level whatever the user preference is. If we double the frame rate (or double the window length), we introduce some temporal scalability and there are two MEPG frames in the window thus two droppable units and two quality levels. If we introduce SNR scalability into MPEG by using SPEG encoding there are four quality levels hence four droppable units per frame.

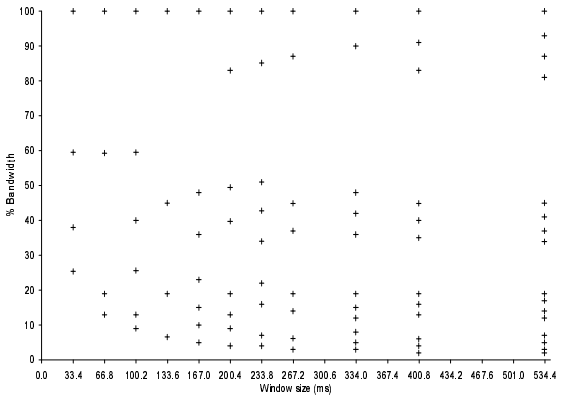


Figure 6. Adaptation granularity vs. window Size (neutral preference)

In order to minimize latency, we need to minimize the window size while maintaining a large enough number of evenly distributed quality levels to enable fine-grain adaptation. For SPEG, a window size of 133.6ms seems to be a good choice, since it has more than 10 quality levels and allows the pipeline to achieve relatively low, less than 400ms, total end to end latency. However, SPEG has only two dimensions of quality adaptation, the temporal adaptation and the SNR adaptation; and the SNR adaptation is relatively coarse-grained. Thus, any results obtained with SPEG could easily be improved with scalable video encodings that provide finer granularity

scalability. With improved scalable video encoding, PPS could easily support interactive streaming with a latency requirement of under 200ms.

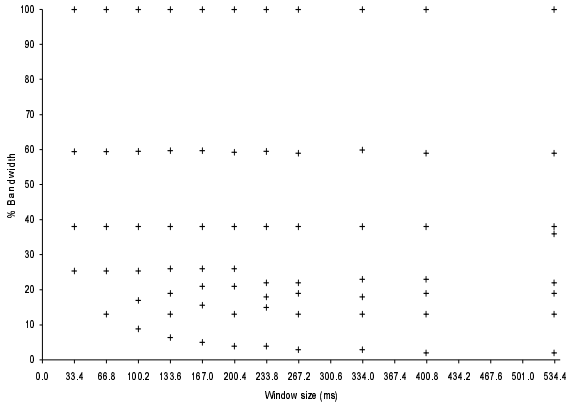


Figure 7. Adaptation granularity vs. window size (prefer temporal quality)

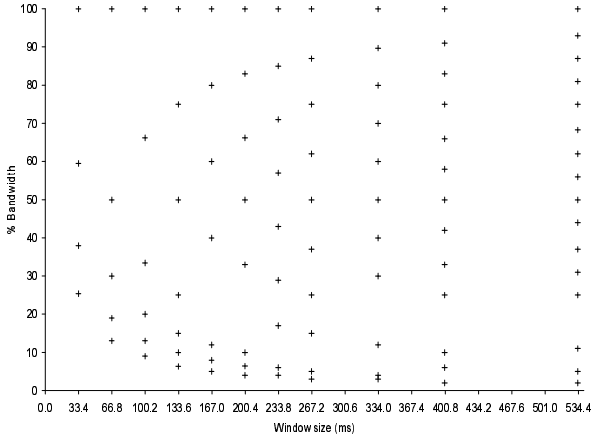


Figure 8. Adaptation granularity vs. window size (prefer SNR quality)

6. Conclusion and future work

Priority progress streaming is a generic and efficient mechanism for fine-grain adaptive streaming of stored media. However, it implies increased latency for reordering data into priority order prior to transmission and for reordering back into time order after transmission. In this paper we explored the real world impact of this reordering latency for live-source video pipelines. We showed that even using a coarse-grained scalable video encoding approaches reordering latency can be reduced to under 400ms, making the approach applicable to many video surveillance applications. As finer granularity video encodings become available, the same level of fine-grain adaptivity will be available using even smaller reordering windows, and interactivity will be easily supported using PPS.

In the future, we plan to extend PPS for multicast delivery and to build a many-to-many video surveillance infrastructure using it.

7. References

[1] J.-C. Bolot and T. Turletti. "Experience with control mechanisms for packet video in the Internet". *ACM SIGCOMM Computer Communication Review*, vol. 28, pp. 4--15, January 1998.

[2] http://ffmpeg.sourceforge.net/

[3] A. Goel and J. Walpole. "Gscope: A Visualization Tool for Time-sensitive Software". In *Proceedings of the Freenix Track of the 2002 UNSENIX Annual Technical Conference*, June 2002.

[4] S. Jacobs and A. Eleftheriadis. "Streaming Video Using Dynamic Rate Shaping and TCP Congestion Control". *Journal of Visual Communication and Image Representation*, 9(3), 211-222, 1998.

[5] C. Krasic and J. Walpole. "QoS Scalability for Streamed Media Delivery". *CSE Technical Report CSE-99-011,* Oregon Graduate Institute, September 1999.

[6] C. Krasic and J. Walpole. "Priority-Progress Streaming for Quality-Adaptive Multimedia". In *Proceedings of the ACM Multimedia Doctoral Symposium*, Ottawa, Canada, October 2001.

[7] C. Krasic, J. Walpole, and W. Feng. "Quality-Adaptive Media Streaming by Priority Drop". To appear in *NOSSDAV 2003*, June 2003.

[8] H. Liu and M.E. Zarki. "Adaptive Source Rate Control for Real-time Wireless Video Transmission". *Mobile Networks and Applications*, 3:49--60, 1998.

[9] R. Rejaie, M. Handley, and D. Estrin. "RAP: An End-to-End Rate-based Congestion Control Mechanism for Realtime Streams in the Internet". In *Proceedings of IEEE INFOCOM*, volume 3, page 1337-1345, New York, NY, March 1999.

[10] B. Vandalore, W. Feng, R.Jain, and S. Fahmy. "A Survey of Application Layer Techniques for Adaptive Streaming of Multimedia". *Journal of Real Time Systems (Special Issue on Adaptive Multimedia)*. January 2000.

[11] D. Wu, Y. T. Hou, W. Zhu, Y-Q Zhang, and J. M. Peha. "Streaming Video over the Internet: Approaches and Directions". *IEEE Transaction on Circuits and Systems for Video Technology*, VOL. 11, NO. 3, March 2001.

APPENDIX K

Quality-Adaptive Media Streaming by Priority Drop. Charles Krasic, Jonathan Walpole, Wu-chi Feng, in Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2003), Monterey, California, June 2003.

Quality Adaptive Media Streaming by Priority Drop*

Charles Krasic, Jonathan Walpole, Wu chi Feng
OGI/OHSU
Beaverton, Oregon
krasic,walpole,wuchi@cse.ogi.edu

ABSTRACT

This paper presents a general design strategy for streaming media applications in best effort computing and networking environments. Our target application is video on demand using personal computers and the Internet. In this scenario, where resource reservations and admission control mechanisms are not generally available, effective streaming must be able to adapt in a responsive and graceful manner. The design strategy we propose is based on a single simple idea, priority data dropping, or *priority drop* for short. We evaluate the efficacy of priority drop as an adaptation tool in the video and networking domains. Our technical contribution with respect to video is to show how to express adaptation policies and how to do *priority-mapping*, an automatic translation from adaptation policies to priority assignments on the basic units of video. For the networking domain, we present *priority-progress* streaming, a real-time best-effort streaming protocol. We have implemented and released a prototype video streaming system that incorporates priority-drop video, priority mapping, and priority-progress streaming. Our system demonstrates a simple *encode once, stream anywhere* model where a single video source can be streamed across a wide range of network bandwidths, on networks saturated with competing traffic, all the while maintaining real-time performance and gracefully adapting quality.

Categories and Subject Descriptors: C.2.2 [Computer Systems Oraganization]: Network Protocols

General Terms: Algorithms, Measurement, Experimentation

Keywords: Quality Adaptive Streaming, Priority Mapping, Internet

*This work was partially supported by DARPA/ITO under the Information Technology Expeditions, Ubiquitous Computing, Quorum, and PCES programs and by Intel.

NOSSDAV'03, June 1–3, 2003, Monterey, California, USA.

1. INTRODUCTION

The Internet is a best-effort environment, where users come and go without resource reservations or admission control. The Internet relies on a model of voluntary cooperative sharing, the foundation of which is congestion control in transport protocols, mainly TCP. It is widely acknowledged that the use of congestion control has been an essential part of the Internet's stability to date [7]. If streaming traffic is to avoid threatening the overall stability, then it too must employ congestion control. Congestion control adapts the sending rate of a flow to share with other flows on the path, changing rates as traffic from other flows comes and goes. As a result, Internet traffic is very bursty. Numerous studies of Internet traffic patterns have shown that traffic rates exhibit significant variation over the full range of time scales, exhibiting so-called self-similar behavior [5, 30]. Given this burstiness, it follows that it is highly unlikely that a single target bitrate will suffice for Internet streaming. If the rate estimate is too conservative, the video stream will under utilize the network, and the resulting video quality will be lower than necessary. On the other hand, if the rate estimate is too aggressive, then the transfer can not complete in real-time and so there will be a streaming failure. For longer duration streaming, the chances are good that a single rate will be too conservative at some times *and* too aggressive at others. Several researchers have recognized these issues, and proposed quality-adaptation instead of single-rate streaming [31].

There have been many proposed techniques for the adaptive delivery of compressed video data over networks. The common idea of quality-adaptive streaming techniques is to adapt dynamically to environmental changes through adjustments in the rate-distortion ratio of the video. Adapting quality over best-effort networks is extremely difficult given the bit-rate changes that occur over time in both.

In this paper, we describe a design strategy for quality-adaptive streaming software. Our strategy revolves around the idea of using priority data dropping, *priority drop* for short, as the primary means of adaptation. In priority drop, the basic data units of the media are explicitly exposed and appropriately prioritized, with the goal that priority-order dropping of data units will yield a graceful reduction in media quality, as we will show in the experimentation section. Our contributions are in two areas, video adaptation and network streaming.

The video component of our system makes compressed video streaming friendly through support of priority drop. We describe a video format, called SPEG (Scalable MPEG),

to illustrate how current video compression techniques can be extended to support priority drop[1]. In contrast to random dropping, which results in unusable video at dropping levels of just a few percent, priority drop is informed and can achieve graceful degradation, over more than an order of magnitude in rate. One of the main questions that arises when considering such a range of target rates is what aspect or aspects of video to degrade? The answer can be influenced by several factors, such as the nature of the content, the nature of the viewing device, the personal preferences of authors, viewers, etc. For example, a sports program might benefit most from preserving fidelity of motion, perhaps at the expense of color fidelity. A user with a PDA may place low relative importance on spatial resolution, compared to a user with a full sized screen.

The scalable coding aspects of SPEG are not our main focus, but rather our main contribution to video entails efficient support for *tailorable* adaptation. We describe a simple method to specify adaptation policies and an associated *priority-mapping* algorithm. The priority-mapping algorithm translates the policy specifications into appropriate priority assignments on data units of priority-drop video. With this approach, video compression is decoupled from the final adaptation, which opens the possibility that each piece of content may be adapted in different ways for different scenarios, with far lower complexity than actually recompressing or transcoding the content during streaming. This added flexibility makes content more re-usable. The ideal streaming-friendly media format would have the characteristics of "encode-once, stream anywhere."

The second component of our contribution is in network streaming. We present an algorithm for real-time best-effort streaming called Priority-Progress streaming (PPS). PPS combines *data re-ordering* and *dropping* to maintain timeliness of streaming in the face of unpredictable throughput. The data units of the priority-drop video are sent in priority order. The algorithm is best effort in that it allows the congestion control mechanism to decide appropriate sending rates. When this sending rate is low, the timeliness of the stream is maintained by dropping low-priority data units at the sender, *before* they would otherwise reach the network. In this way, the amount of higher-priority data sent automatically matches the rate decisions of the congestion-control mechanism.

We have implemented a streaming system which integrates SPEG, priority-mapping, and PPS, with the following results. First and foremost, it maintains timeliness of the stream in the face of rate fluctuations in the network. Second, PPS makes full use of available bandwidth, and achieves full goodput from that bandwidth, thereby maximizing the average video quality. The bandwidth used is limited by the congestion control of the underlying transport, in our case TCP, so the usage represents a fair share, in friendly consideration of the Internet's existing traffic mix [7]. Third, it starts quickly when the user initiates the stream, avoiding a long pre-buffering period. Finally, it limits the number of quality changes that occur, by using bandwidth skimming to increase client-side buffering concurrent during normal playout. The overall message of our results is that priority drop is very effective: a single video can be streamed across a wide range of network bandwidths, on networks heavily saturated with competing traffic, while maintaining real-time performance and gracefully adapting quality.

The remainder of this paper is organized as follows. In the next section we discuss problem background and related work. In Section 3, we describe priority-drop video and priority mapping. In Section 4 we describe the details of the PPS algorithm. Section 5 presents experiments and results from our prototype streaming video system. Finally, discussion and conclusions are in Section 6.

2. BACKGROUND

Most streaming content on the Internet today is provided using one of three streaming platforms: Microsoft's Windows Media, Real Networks RealSystem, and Apple's QuickTime. To various degrees, these systems adhere to a suite of standards related to streaming such as RTP, RSTP, SIP, and SMiL. [9, 8, 26, 27]. Quality adaptation algorithms are outside the scope of any of these standards. In particular, while adaptation might be layered on RTP, RTP does not provide any direct algorithm for quality adaptation. These commercial systems all employ proprietary quality adaptation. Although these proprietary adaptation mechanisms are largely secret, we make some high level observations in the following paragraphs based on published information.

In addition to commercial activity, there has been extensive academic research related to video streaming over the Internet. The research spans several distinct domains, including video compression, real-time systems, and networking. A spectrum of adaptive strategies have been proposed to deal with the consequences of best effort service [31]. One of the most commonly used strategies is *one-time adaptation*, where the user chooses between a small set of predetermined rates before streaming begins. Once started, streaming is fixed at this single-rate regardless of competing traffic, hence this approach retains the basic problems of single-rate streaming mentioned earlier, where it is prone to yield lower quality than necessary when more bandwidth is available and prone to complete failure when less bandwidth is available. Both problems are more probable for longer duration content. Apple's Quicktime uses one-time adaptation and in addition it adjusts the amount of client-side buffering based on measured rate volatility during startup [29]. Startup time, while initial buffering is established, can be quite high—on the order of tens of seconds. Windows Media and RealSystem based systems are often configured in this mode also, even though they do support more advanced mechanisms, which we'll describe below. In the remainder of this section, we expand on the basic performance issues for quality-adaptive streaming, in light of some of the approaches proposed in the literature and in terms of the commercial streaming systems.

The related work to this paper falls into four main categories. *Multi-version* techniques store a single video at a range of pre-selected bitrates (e.g. Windows Media IntelliStream and Real's SureStream) [4, 1]. While simple to implement, multi-version supports only coarse adaptation and under utilizes storage. *Online scaling* techniques support changing the target rate parameter of the encoder or the transcoder on the fly. While these support fine-grained adaptation, the computational time required to recode limits scalability of these approaches. *Scalable video coding* technologies focus on creating compression formats that allow

[1]SPEG is similar to MPEG FGS, but easier to implement with publicly available software

adaptation of the rate-distortion relationship without explicitly re-coding (e.g. MPEG-2 scalability, MPEG-4 FGS). These techniques are complementary to the work we describe here. Advances in these areas can be directly incorporated into our framework. While the first three categories are concerned mainly with video representation and coding, the fourth category is *adaptive streaming* which concerns the mechanics of actual network delivery.

Ideally, a quality-adaptive streaming system will select video quality to match the average available network bandwidth. In practice, adaptation tends to be limited to discrete steps, and consequently the rate match is only approximate. A system that supports steps with finer-granularity generally results in a better match, which manifests itself in higher quality and better reliability of streaming. The type of video compression, especially whether the compression is scalable or not, is a major factor influencing the granularity of quality-adaptive streaming.

Because many of the compression formats in common use are not explicitly scalable [15, 14, 16, 13], the target rate is a required parameter for encoding. These formats do not provide explicit support for adapting rate after encoding. Frame dropping is a well known work-around, and is probably the most popular video adaptation mechanism, having been used since the first quality-adaptive Internet streaming systems appeared [2].

Online-scaling techniques, which include live encoding, transcoding, and data-rate shaping (DRS), allow changing the target rate parameter of the encoder or transcoder on the fly [17, 32]. Transcoding and DRS can have significantly lower computational complexity than encoding. The main advantage of online scaling is very fine granularity. However, even the most efficient DRS is very computationally intensive relative to non-adaptive streaming, or adaptive streaming through frame dropping or multi-coding. This extra computational cost poses a major obstacle to supporting very large numbers of independently adaptable streams in servers and edge devices.

In contrast, scalable compression aims to support low-complexity adaptation that will scale to large numbers of streams. Scalable compression schemes explicitly support multiple quality levels, exposing two or more layers in the encoded video. The layers are progressive, the higher layers depend on the lower layers, and the higher layers are used to refine quality. The various scalable compression approaches differ in terms of granularity, ranging from very coarse, as in the work in Layered Multicast [23] and MPEG-2 Scalability [10], to very fine, such as in recent work in MPEG-4 and H.26L Fine Granularity Scalability [22, 11]. With the current state of the art, scalable video compression comes with a compression efficiency penalty, in that video quality is lower compared to the results of non-scalable compression at the same rate, but this penalty is getting smaller [11]. Fine granularity scalability through layering makes it possible to begin streaming without even knowing the target rate, by sending lower layers before higher layers and truncating higher layers if time runs out. Contrast this approach with online-scaling, where the quality adaptation must commit to a target rate *before* encoded data is ready to transmit. In exchange for the small efficiency penalty, scalable compression offers a significant boost in freedom for the design of adaptive streaming mechanisms.

A principal concern with streaming is the potential impact of video traffic on existing Internet traffic. Many research projects have studied quality adaptive streaming in relationship with *TCP-friendly* congestion control [31, 25, 3, 6, 17, 28, 19]. A common idea among them is to let the transport protocol and its congestion control dictate the appropriate sending rate. The main differences are in the details of deciding what to send and what to drop, and what information are used to inform these control decisions. For example, Rejaie *et al* describe their algorithms for optimal streaming [25], where optimal means minimal client-side buffering, and thus a minimal associated contribution to end-to-end latency. The role of their algorithm is to control adding and removing quality layers, where the control decisions are based on a rate-driven feedback control. The design of their control is based on analysis of additive-increase multiplicative-decrease (AIMD) congestion control[2] and an assumption of apriori knowledge of video rate requirements [25]. *Feamster et al* extend this work to more general congestion control mechanisms [6]. In contrast to these systems that explicitly attempt to match rates, *Feng et al* describe an adaptive streaming algorithm that uses a sliding window over video frames, sending data from low to high quality, in best effort fashion [3]. Feng's algorithm gains simplicity because it does not attempt to absolutely minimize client-side buffering, and has the advantage of working without direct assumptions about the design of the underlying congestion control. Kang et al. [18] propose a priority-driven adaptation, but assuming fixed bandwidth channels. The question of how to link scalable video encoding and tailorable adaptation policies to TCP-friendly streaming is open, and is the topic of this paper.

We use scalable compression and TCP in this paper. One of the contributions of our approach is to demonstrate the benefits of using the priority-timestamp packet as the basic unit of media abstraction, as opposed to video frames, or layers in a stream. Through priority-mapping, we extend scalable video compression to support tailorable adaptation, so that compromises made in quality better reflect the influence of specific content, viewing devices, and user preferences. Our Priority-Progress Streaming algorithm extends TCP-friendly adaptive streaming to support direct control over quality compromises in streaming, such as latency limits, and limits on the number of quality changes, while preserving the goals of high utilization and video quality.

3. STREAMING-FRIENDLY VIDEO

In this section we will describe how scalable video compression can support tailorable adaptation through priority drop. This consists of a scalable video format and a *Priority Mapper*. We have implemented an adaptive streaming system based on our approach, called the Quasar Video Pipeline. In lieu of a freely available implementation of the more recent scalable compression systems [22, 11], we have developed a minimal scalable compression format we call SPEG (Scalable MPEG), derived from MPEG-1 video. Our purpose in implementing SPEG was to test priority mapping and PPS using real video. Priority mapping is the main subject of this section, but we first give a brief description of SPEG for the benefit readers not familiar with scalable compression formats such as MPEG-4 FGS.

[2]TCP's congestion control uses an instance of AIMD after it reaches steady state.

3.1 Scalable Video

In MPEG video, each frame is broken down into 8x8 pixel blocks, which are converted to corresponding 8x8 blocks of coefficients using the discrete-cosine transform (DCT). *Quantization*, strategic removal of low order bits from these coefficients, is the primary basis for compression gains in MPEG and very many other similar compression schemes. SPEG transcodes MPEG coefficients to a set of levels, one base level and three enhancement levels as follows. If we denote the original MPEG coefficients $X[i,j]$, then SPEG partitions this coefficient data according to the following equations[3]:

$$\begin{aligned} X_{base}[i,j] &= X[i,j] >> 3 \\ X_{e0}[i,j] &= (X[i,j] >> 2) \;\&\; 1 \\ X_{e1}[i,j] &= (X[i,j] >> 1) \;\&\; 1 \\ X_{e2}[i,j] &= X[i,j] \;\&\; 1 \end{aligned}$$

The coefficients from each level are grouped to form layers, four per original MPEG frame, which are the basic *application level data units* (ADUs) in SPEG. The above steps can be reversed to return SPEG back to the original MPEG. Alternatively, we can drop some or or all of the enhancement layer ADUs (from high to low) substituting zero values for the missing data. The effect of such dropping is analogous to having used higher quantization parameters during MPEG encoding, yielding lower bitrate in exchange for less spatial fidelity. We present SPEG because it suffices to demonstrate the essential properties of scalable compression and because it is readily available to us. Our techniques would apply to most scalable formats, e.g. MPEG-4 FGS.

We expect future scalable codecs will expose even more scalability mechanisms. One example is *spatial-size scalability*, where the number of pixels of height and width are scalable. Another example is *chroma scalability* which might allow a range of color fidelities, from 4:4:4 to 4:2:2 to 4:1:1 to greyscale to monochrome. The object based compression techniques might allow *content adaptation* through addition and removal of objects[16]. These possibilities raise the issue of tailorable adaptation. In order to take full advantage of all of these scalability options, there would need to be a good way to control how they are used together. To explore tailorable adaptation, we use SPEG's spatial scalability in combination with frame dropping to provide a minimal example of a compression scheme with more than one scalability mechanism.

3.2 Priority Mapping

Having more than one quality dimension leads to the issue that choosing how to best adapt the multiple dimensions may depend on the usage scenario. For example, the target device may have a small screen, so preserving frame-rate may make more sense than spatial detail. A user may want to repeat a scene in slow motion, which looks smoother if more frames are inserted. Conversely, skipping frames is harder to notice when doing fast-forward scan. We have designed a priority-mapper with the intent of providing a general and flexible approach to tailoring quality adaptation to such specific quality preferences. The priority-mapper automatically assigns priorities to the units of a media stream, so that priority drop yields the most graceful degradation, as appropriate to the viewing scenario.

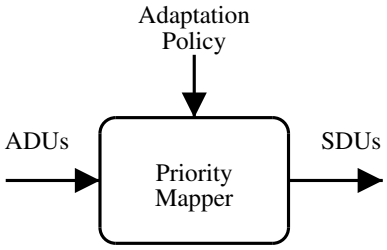


Figure 1: Priority Mapper

Figure 1 depicts the mapper used in the Quasar Video Pipeline. The mapper's inputs are *application data units (ADUs)* and the quality adaptation policy. The mapper's output are *streaming data units (SDUs)* which are aggregates of prioritized ADUs, where the aggregation is based on ADUs which have the same priority and timestamp value. The purpose of the aggregation is to isolate the PPS algorithm from low level details of the video format, particularly the data dependencies that exist between ADUs.

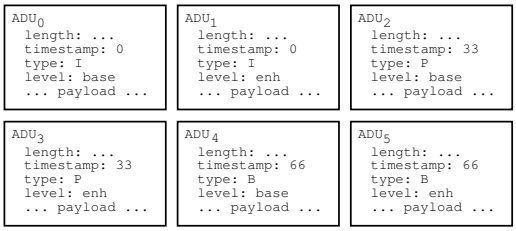


Figure 2: ADUs

Figure 2 shows a sequence of ADUs. The ADUs have a packet like form, consisting of a fixed-length header, and a variable length payload. The header contains basic information needed by the mapper, such as the length of the payload, a timestamp, and payload specific flags. For example, with SPEG these flags indicate the type of MPEG frame the ADU is part of (I, B, or P), and to which spatial scalability layer the ADU belongs[4].

3.2.1 Specification of Adaptation Policies

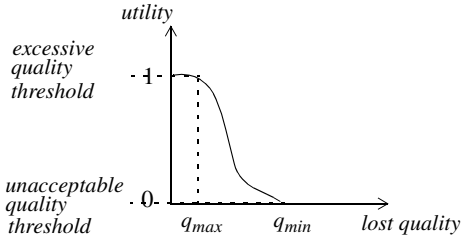


Figure 3: A utility function with thresholds

[3]The >> denotes the right bitwise shift operator, and the & denotes the bitwise *and* operation.

[4]To simplify our examples, figure 2 depicts only two spatial layers, although our SPEG implementation has four.

We use *utility functions* as declarative specifications for the adaptation policy. A utility function is a simple and general means for users to specify their preferences. Figure 3 depicts the general form of a utility function. The horizontal axis describes an objective measure of lost quality, while the vertical axis describes the subjective utility of a presentation at each quality level. The region between the q_{max} and q_{min} thresholds is where a presentation is acceptable. The q_{max} threshold marks the point where lost quality is so small that the user considers the presentation "as good as perfect." The area to the left of this threshold, even if technically feasible, brings no additional value to the user. The rightmost threshold q_{min} demarks the point where lost quality has exceeded what the user can tolerate, and the presentation is no longer of any use. The utility levels on the vertical axis are normalized so that zero and one correspond to the "useless" and "as good as perfect" thresholds. In the acceptable region of the presentation, the utility function should be continuous and monotonically decreasing, reflecting the notion that decreased quality should correspond to decreased utility. In the case of priority mapping for SPEG, the adaptation policy consists of two utility functions, one for spatial quality and one for temporal quality.

3.2.2 Automatic Translation from Policy to Priorities

The mapping algorithm subdivides the timeline of the media stream into intervals called *mapping windows*. The size of the interval is a parameter to the mapping algorithm, but may be adjusted in order to meet alignment requirements; for example, the mapping window is a sequence of one or more complete GOPs for SPEG. The mapping algorithm prioritizes the ADUs within each window separately. For all ADUs in a given mapping window, the mapping algorithm finds the order in which ADUs may be dropped that has the minimum impact, given the data dependency rules of the video and the preferences specified via the given utility functions. The final priority assignment will be used by the streaming algorithm to guide quality adaptations, while accurately reflecting user preferences.

We use the ADUs from figure 2 as an example mapping window, which consists of a single GOP and spans the interval 0–66 ms. The priority mapping algorithm processes the ADUs within a window in two phases.

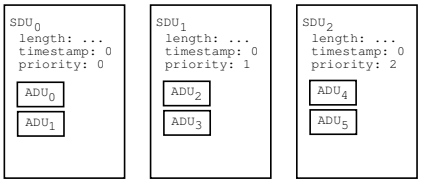
In the first phase, the ADUs are partially ordered, according to a *drop before* relationship[5], based on video data dependencies. For example, the spatial layering requires that base layer ADUs should not be dropped before their corresponding enhancement layer ADUs, which applies to the ADUs of figure 2 as follows:

$$ADU_1 => ADU_0 \quad ADU_3 => ADU_2 \quad ADU_5 => ADU_4$$

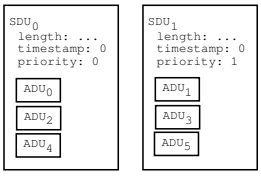
Similarly, MPEG's predictive coding rules (for I,P,B frames) are expressed as follows:

$$ADU_4 => ADU_2 => ADU_0$$

These first two sets of ordering constraints represent *hard dependency* rules, in that they simply reflect SPEG semantics. The mapper adds some other *soft dependency* rules which improve adaptation results. With video, for example, the mapper would add soft-dependencies so to ensure that frame dropping be as evenly spaced as possible[6]. After the first mapping phase, there still remains significant freedom for adaptation. For example, figure 4 contains two very different mappings for the ADUs of figure 2, yet both mappings adhere to the phase one constraints above.



(a) Frame drop



(b) Spatial drop

Figure 4: SDUs: prioritized and grouped ADUs

The second phase of the priority mapper algorithm is where the adaptation policy is used to refine the partial ordering from the first phase, generating the (totally ordered) prioritized SDUs.

The algorithm works through an iterative process of elimination over the ADUs. We say an ADU is *alive* if it is still in the set of unprioritized ADUs, and *dead* otherwise. Each iteration considers the set of candidate ADUs which are not yet dead, initially all ADUs from the mapping window, and have no living dependents, based on the constraints generated by the first phase. For each of these candidate ADUs, and each quality dimension (spatial and temporal in SPEG), the mapper computes the presentation quality would result if the candidate ADU were dropped, that is, the quality is computed based on all ADUs that are still alive, less the current candidate. For the temporal quality dimension, the mapper computes the frame rate, and for spatial quality the spatial level. At this point the mapper is ready to apply the adaptation policy. The utility functions are used directly to convert the computed quality values to corresponding utilities. The "overall utility" for each ADU is just the *minimum* of its per dimension utilities. The candidate ADU that has the highest utility is selected as the next victim; i.e. dropping this ADU next has the smallest impact on utility. The priority value for the victim ADU is a linear (inverse) fitting of the utility into the range of priority values. For example, in the Quasar pipeline this fit goes from a utility range of 0 to 1 to a priority range of 15 to 0[7]. The iterations stop when all ADUs have been assigned a priority.

Once all the ADUs have priorities, they are then grouped into SDUs, one per priority level. The SDUs are all set to

[5]This is really drop no-later than, since dropping is always optional.

[6]If half the frames are to be dropped, then it is best to drop every other frame, as opposed to more clustered dropping such as keeping even GOPs and dropping odd GOPs

[7]Maximum priority is 15

Video	Resolution	Length (frames)	GOP length
Giro d'Italia	352x240	1260	15
Wallice and Grommit	240x176	756	3
Jackie Chan	720x480	2437	8
Apollo 13	720x480	864	6
Phantom Menace	352x240	4416	16

Figure 5: Movie Inputs. The movies were coded with several different MPEG encoders. A variety of content types, movie resolutions, and GOP patterns were chosen to verify our techniques perform consistently.

have the timestamp of the first ADU in the window. This grouping simplifies matters for later stages, like the PPS algorithm and the video decoder[8].

3.3 Mapping Results

We now present some the results of mapping for several test movies. Figure 5 describes the set of movies used, which were prepared with a variety of encoders and encoder parameters. In figure 6(a) and (b) we set a quality adaptation policy consisting of equal linear utility functions for temporal and spatial quality. Figures 6(c) and (d), show the priority-assignment produced by the mapper. At each threshold, the quality corresponds to when all packets with priority lower than the threshold are dropped. For example, at priority threshold 6, 20 fps is achieved at SNR level 3.

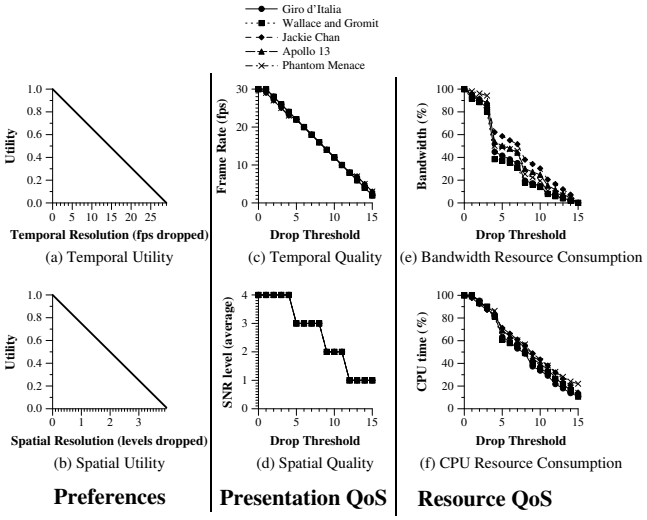


Figure 6: QoS Mapping Applied to SPEG

Ideally, the presentation quality graphs would look the same as the utility functions they were derived from. In particular, the range of acceptable presentation QoS would be covered, and the shape of adaptation would follow the shapes of the utility functions. Figure 6(c) shows the relationship between presentation-QoS for temporal resolution (frame rate) and priority-drop threshold. It should be noted that figure 6(c) contains lines for each of the test movies, but they overlap very closely because the mapper is able to label packets to follow the utility function policy closely. Although desirable, this result was not entirely expected because MPEG's inter-frame dependencies constrain the order in which frames can be dropped, and some GOP patterns are particularly poorly suited to frame dropping. On the spatial resolution side, in figure 6(d), we note that the mapper drops resolution levels uniformly across all frames, resulting in a stair-shaped graph, since there are only 4 SNR levels in SPEG. In as much as the SPEG format allows, the presentation-QoS matches the specified user preferences.

The resource side of the adaptation profiles are shown in the third pair of graphs in Figures 6(e) and (f). We show the average bandwidth of the movies at each drop threshold, as a percentage of the bandwidth when no packets are dropped. Similarly, we show the CPU time required for client side processing (decoding) of the video at each drop threshold, where the values are normalized to the CPU cost when no packets are dropped. A good shape for these graphs would be smooth and linear over a wide range of resource levels. We see that bandwidth in Figure 6(e) does indeed range all the way down to only a few percent. What this means is that the quality-mapper can prioritize the video to operate in extremely diverse networking and computing environments. CPU time in Figure 6(f) is very nice and smooth, although it does not cover as much range as bandwidth, and reaches a minimum of about 10 percent. We also note that the movies are closely clustered in their resource-QoS graphs, indicating that adaptation is independent from differences in encoders or encoder parameters. Further results for other policies are presented in [21].

4. PRIORITY PROGRESS STREAMING

In this section, we present an overview of the PPS algorithm. While the priority-drop video encoding and the priority-mapper described in the previous section do a substantial amount of preparation for delivery, the streaming algorithm still plays a key role in realizing the benefits of adaptive streaming.

The objective of our streaming algorithm is to take the SDUs produced by the Priority Mapper, and using their timestamp and priority labels, perform real-time adaptive streaming over a TCP-friendly transport. As it happens, our implementation of the algorithm works quite well over an unmodified TCP protocol.

The PPS algorithm works by subdividing the timeline of the video into disjoint intervals called *adaptation windows*. Adaptation windows are distinct from the mapper windows described in the previous section, an adaptation window consists of one or more mapper windows.

Figure 7 shows the conceptual outline of Priority-Progress Streaming. A pair of re-ordering buffers is employed around a *bottleneck*, which in our case is the TCP session. The buffers contain the SDUs of an adaptation window. The algorithm for Priority-Progress Streaming contains three subcomponents, the upstream buffer, downstream buffer, and progress regulator respectively. The upstream buffer admits all SDUs within the time boundaries of an adaptation window, these boundaries are chosen by the progress regulator. Each time the regulator advances the window forward, the

[8]Otherwise there can be pathological cases during streaming where low priority ADUs for one timestamp are kept even though higher priority ADUs with different timestamps, but belonging to the same mapping window, are dropped. For example, a P frame (low priority) might be kept when it's I frame (high priority) was dropped, however the P frame can not be decoded properly without the dropped I frame.

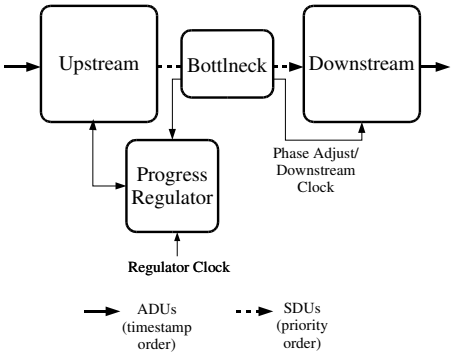


Figure 7: Priority-Progress Conceptual Architecture

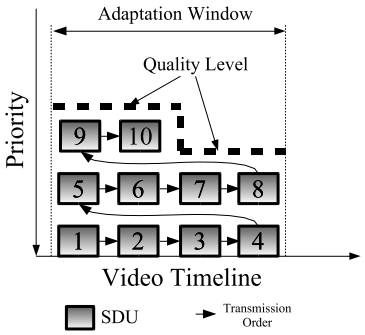


Figure 8: Adaptation Window Transmission: There are at most two quality levels per window.

unsent SDUs from the old window position are expired and the window is populated with SDUs of the new position. SDUs flow from the buffer in priority-order through the bottleneck to the downstream adaptation buffer, as fast as the bottleneck will allow. In order to sort into priority order, the buffer is implemented via a priority queue data structure. Similarly, the downstream adaptation buffer collects SDUs and re-orders them to timestamp order. When SDUs arrive late because of unexpected delays through the bottleneck, the progress regulator is notified so that it may avoid late SDUs in the future. The downstream buffer receives as many SDUs as the bandwidth of the bottleneck will allow and the rest, which are of lowest priority, are dropped at the server. In this way, the dropping will adapt video quality to match the network conditions between the sender and the receiver.

Window Number	Prepare		Transmit		Display	
	Start	End	Start	End	Start	End
1	0	1	1	2	2	3
2	1	2	2	3	3	4
3	2	3	3	4	4	5
4	3	4	4	5	5	6
5	4	5	5	6	6	7

Table 1: Priority Progress Example

As described in the paragraph above, each adaptation window goes through three distinct processing phases. The first phase is *window preparation*, which includes retrieval from the source (file or live capture), prioritization, and re-ordering from timestamp to priority order. The second phase is *window transmission*, where the SDUs are transmitted in priority order. The third phase is *decoding and display*. Table 1 gives a simple example for a sequence of five adaptation windows, where each row describes the timing of the phases for the nth adaptation window.

4.1 Responsiveness and Consistency

The basic premise of streaming is to start display as soon as possible relative to the start of transmission. Quality adaptive streaming has the added objective to adjust video quality so as to make full use of the available bandwidth, both to increase average quality and to prevent long term rate fluctuations from disrupting the stream entirely. However, it is also true that it is preferable to avoid exposing the user to visible quality changes. The size of adaptation windows in PPS determine important trade-offs between streaming latency, buffer space requirements, robustness to rate changes, and the consistency of quality. Smaller windows have the advantage of shorter startup delay, because the algorithm does not allow display of a window until transmission is fully completed. Larger windows have the advantage that quality will change less often, and larger rate fluctuations can be smoothed out.

In Priority-Progress, the sizes of the adaptation windows have a direct effect on the number of quality changes. Figure 8 shows how the final quality level, for a given adaptation window, is determined by the transmission order used in Priority Progress. The SDUs for the window are transmitted primarily in priority-order, and secondarily in timestamp order, as in the figure. So the transmission pattern is like filling the rectangle from left to right, bottom to top. In the end, there are (upto) two priority levels that have been reached, hence two quality levels, as shown by the dashed line[9]. Then in the limit, the total number of changes for the whole video is two times the number of adaptation windows in the video timeline. In this way, longer adaptation windows directly ensure more consistent quality, in that longer windows decrease the number of possible quality changes.

4.2 Window Scaling

The fact that shorter and longer adaptation windows each have their benefits reflects what is likely an inherent trade-off between responsiveness and consistency in adaptive streaming. However, it is not necessary to restrict all window sizes to the same value. The Priority-Progress algorithm includes the option to adjust the window size during the streaming process, which we call *window scaling*. With window scaling, the window duration starts out minimal, so that startup latency is minimal, and then the window duration grows with each new window as the stream plays. As the window durations get larger, the quality changes become less frequent. Compared to a fixed window duration, we will see that window scaling yields dramatically better balance between responsiveness and consistency.

Window scaling is possible because Priority Progress can transmit the video at a faster (or slower) rate than it will

[9]This assumes that quality for a single priority level is uniform, which is true for our priority mapper algorithm.

Window Number	Prepare			Transmit			Display		
	Duration	Start	end	Duration	Start	End	Duration	Start	End
1	1	0	1	0.5	1	1.5	1	1.5	2.5
2	2	1	3	1	1.5	2.5	2	2.5	4.5
3	4	3	7	2	2.5	4.5	4	4.5	8.5
4	8	7	15	4	4.5	8.5	8	8.5	16.5
5	16	15	31	8	8.5	16.5	16	16.5	32.5

Table 2: Window Scaling Example: windows grow at 100% rate

be consumed at the receiver. The consumption rate at the receiver is naturally fixed to the videos "real time" rate, but transmission schedule is not so constrained. The priority dropping mechanism is what affords flexibility in this respect. Sending a window faster just means that more SDUs may be dropped. In altering the transmission schedule, the Priority Progress algorithm can create (or reclaim) *workahead* in the streaming schedule, which is what allows subsequent adaptation windows to be larger (or smaller). Workahead accumulates whenever the duration of the transmission phase is shorter than the display phase. By definition, the transmission of the first adaptation window is a preroll window which establishes the initial workahead. With the exception of the preroll window, the accumulated workahead is the upper bound on duration of each step of the transmission phase. We call the ratio between duration of a transmission phase step and the duration of the corresponding display phase step the window scaling *growth ratio*.

Table 2 describes a timeline for five adaptation windows, where the growth ratio is fixed at 2. As in table 1, each row in the table describes processing for a single adaptation window. The columns show when the timing of the three phases for each window. We use a growth ratio of 2 here as it results in relatively simple numbers, but in practice we use more modest ratios. The results in the next section are based on a ratio of 1.1.

4.3 Priority-Progress Streaming Results

In this section, we describe experiments and results for PPS using our Quasar pipeline implementation. Our experimental setup consists of a group of Linux based PCs acting both as end hosts and as a router in a dedicated network testbed that implements a saturated network path. The router runs the NISTNet wide area network emulation package [24], which allows us to introduce artificial delay and bandwidth limitations. For the experiments presented here, we set the delay to produce a 50ms round-trip-time. We also set a bandwidth limitation of approximately 25Mbps and impose a queue length limit that matches the bandwidth delay product. For the entire duration of the experiments, the network is *saturated* with competing traffic.

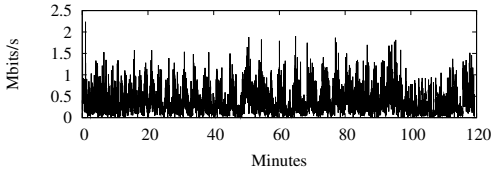
We have written a synthetic traffic generator, called `mxtraf` [20], that we use to generate the various levels and mixes of competing traffic. The mix is made up of non-responsive UDP traffic (10%), short-lived (20Kb) TCP flows (˜60%), and long-lived infinite-source TCP flows (˜30%), similar to measurements reported in [12] . Our experiments consist of streaming a two hour video through this saturated network path. To provide baseline performance references, we simulate two existing streaming algorithms assuming they are given the same video and available bandwidth from our experiments. The first algorithm is based on the Berkeley CMT, and the second on Feng's technique[3]. We then show the performance of PPS in two cases, the first using a fixed adaptation window, and the second with the PPS adaptation window scaling feature enabled.

Figure 9(a) shows the transmission rate of the TCP session used to transport the video. Figure 9(b) shows the maximum rate requirement of the video, which is significantly above the rate achieved by our TCP stream in the given conditions. For each streaming algorithm we show the frame-rate and SNR level achieved over the course of the whole stream[10]. Figures 9(c) and 9(d) show that the CMT algorithm has great difficulty with the conditions of our experiment. Video quality is extremely volatile, and there are several instances where the algorithm is not able to deliver even the minimum quality. Figures 9(e) and 9(f) show the sliding window algorithm fares much better, with fewer quality changes and no failures. Figures 9(g) and 9(h) show PPS with a fixed adaptation window behaves quite similarly to the sliding window approach. It would be possible to improve the consistency of PPS in the fixed window case by increasing the size of the window, but that would come at the direct expense of startup latency. The major benefits of PPS arise the adaptive window scaling is enabled, shown in figures 9(g) and 9(h), where quality gets more consistent over the course of the stream. In the majority of the movie, quality changes are several minutes apart, even though startup latency is in the range of 1 second.

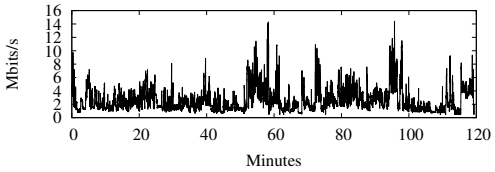
5. CONCLUSIONS AND FUTURE WORK

Streaming video over the Internet remains a compelling and challenging problem. While video compression addresses the issue of limited bandwidth, it is only recently that scalable compression has addressed the extra problem of highly variable bandwidth as on the Internet. Also recently, there has been consensus that video traffic should employ TCP friendly congestion control if it is to avoid threatening existing traffic and stability. In this paper, we presented a framework for adaptive video streaming centered around the simple concept of priority drop. We showed how, through priority-drop, to combine scalable compression and adaptive streaming in to form a very effective, tailorable, adaptive streaming system, supporting an encode-once, stream anywhere model. For future work, we are considering several extensions of the Quasar pipeline, including incorporating Priority-Progress streaming to an Application Level Multicast Overlay, extending Priority-Progress to inter-stream adaptation, and incorporating video compression with better and more scalability options.

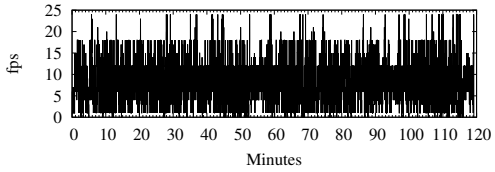
[10]Recall SPEG has four SNR levels



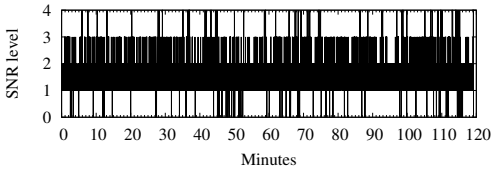
(a) Video stream TCP Transmission Rate (smoothed to 1s intervals)



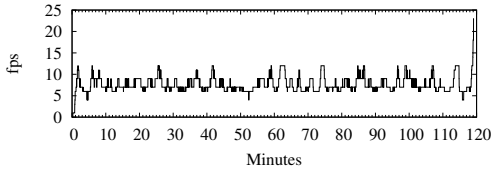
(b) Maximum Video Rate



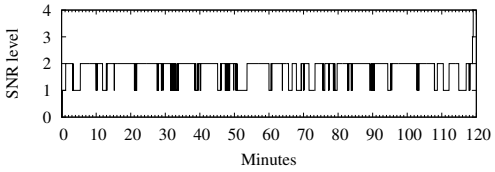
(c) CMT (2s buffer)



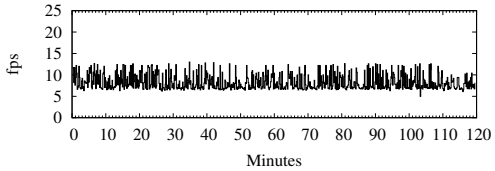
(d) CMT (2s buffer)



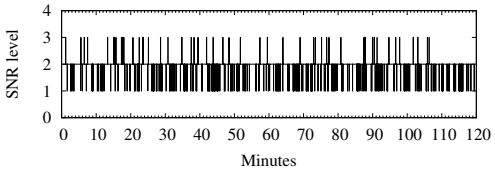
(e) Sliding Window Smoothing (60s window)



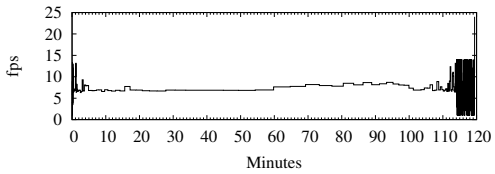
(f) Sliding Window Smoothing (60s window)



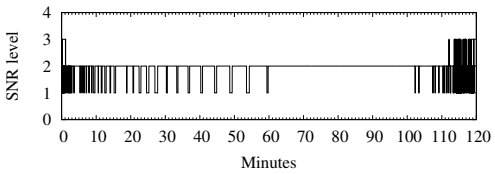
(g) PPS (10s window fixed)



(h) PPS (10s window fixed)



(i) PPS with adaptive window scaling (10%)



(j) PPS with adaptive window scaling (10%)

Figure 9: Sub-figure (a) shows the transmission rate in a saturated network over a two hour period. Sub-figure (b) shows the maximum rate of the video. Sub-figures (c)-(j) show the resulting video quality with each of four streaming algorithms.

6. REFERENCES

[1] B. Birney. Intelligent Streaming. http://msdn.microsoft.com/, October 2000.

[2] S. Cen, C. Pu, R. Staehli, C. Cowan, and J. Walpole. A Distributed Real-Time MPEG Video Audio Player. In *Network and Operating System Support for Digital Audio and Video*, pages 142–153, 1995.

[3] W. chi Feng, M. Liu, B. Krishnaswami, and A. Prabhudev. A Priority-Based Technique for the Best-Effort Delivery of Stored Video. In *SPIE/IS&T Multimedia Computing and Networking 1999*, San Jose, California, January 1999.

[4] G. Conklin, G. Greenbaum, K. Lillevold, and A. Lippman. Video Coding for Streaming Media Delivery on the Internet. *IEEE Transactions on Circuits and Systems for Video Technology*, 11(3), March 2001.

[5] M. E. Crovella and A. Bestavros. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[6] N. Feamster, D. Bansal, and H. Balakrishnan. On the Interactions Between Layered Quality Adaptation and Congestion Control for Streaming Video. In *11th International Packet Video Workshop (PV2001)*, Kyongiu, Korea, April 2001.

[7] S. Floyd and K. Fall. Promoting the Use of End-to-End Congestion Control in the Internet. *IEEE/ACM Transactions on Networking*, August 1999.

[8] S. M. W. Group. Synchronized Multimedia Integration Language (SMIL) 1.0 Specification. Technical report, World Wide Web Consortium, 1998. http://www.w3.org/TR/REC-smil.

[9] M. Handley, H. Schulzrinne, E. Schooler, and J. Rosenberg. SIP: Session Initiation Protocol. RFC 2543, March 1999.

[10] B. G. Haskell, A. Puri, and A. N. Netravali. *Digital Video: An Introduction to MPEG-2*, chapter 9. Chapman & Hall, 1997.

[11] Y. He, F. Wu, S. Li, Y. Zhong, and S. Yang. H.26l-based fine granularity scalable video coding. In *ISCAS*, 2002.

[12] G. Iannaccone, M. May, and C. Diot. Aggregate Traffic Performance with Active Queue Management and Drop from Tail. *Computer Communication Review*, 31(3), July 2001.

[13] IEC. 61834 Helical-scan digital video cassette recording system using 6,35 mm magnetic tape for consumer use (525-60, 625-50, 1125-60 and 1250-50 systems). International Standard, 1999.

[14] ISO/IEC. 13818-2 Information technology — Generic coding of moving pictures and associated audio information: Video . International Standard, 1993.

[15] ISO/IEC. 11172-2 Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s — Part 2: Video. International Standard, 1994.

[16] ISO/IEC. 14496-2 Information technology — Coding of audio-visual objects — Part 2: Visual. International Standard, December 1999. First edition.

[17] S. Jacobs and A. Eleftheriadis. Streaming Video using Dynamic Rate Shaping and TCP Flow Control. *Visual Communication and Image Representation Journal*, January 1998. (invited paper).

[18] S. H. Kang and A. Zakhor. Packet Scheduling Algorithm for Wireless Video Streaming. In *Packet Video 2002*, Pittsburgh, April 2002.

[19] J.-W. Kim, Y.-G. Kim, T.-Y. K. H.-J. Song, Y.-J. Chung, and C.-C. J. Kuo. TCP-friendly Internet Video Streaming employing Variable Frame-rate Encoding and Interpolation. *IEEE Transaction on CSVT*, 10, October 2000.

[20] C. Krasic, A. Goel, and K. Li. The MxTraf Network Traffic Generator. `http://mxtraf.sf.net/`.

[21] C. Krasic and J. Walpole. QoS scalability for streamed media delivery. CSE Technical Report CSE-99-011, Oregon Graduate Institute, September 1999.

[22] W. Li, F. Ling, and X. Chen. Fine Granularity Scalability in MPEG-4 for Streaming Video. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS 2000)*, Geneva, Switzerland, May 2000. IEEE.

[23] S. McCanne, M. Vetterli, and V. Jacobson. Low-Complexity Video Coding for Receiver-driven Layered Multicast. *IEEE Journal on Selected Areas in Communications*, 16(6):983–1001, August 1997.

[24] NIST. The NIST Network Emulation Tool. `http://www.antd.nist.gov/itg/nistnet`.

[25] R. Rejaie, M. Handley, and D. Estrin. Quality Adaptation for Congestion Controlled Video Playback over the Internet. In *Proceedings of ACM SIGCOMM '99 Conference*, Cambridge, MA, October 1999.

[26] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 1889, January 1996.

[27] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326, April 1998.

[28] D. Sisalem and H. Schulzrinne. The Loss-Delay Based Adjustment Algorithm: A TCP-Friendly Adaptation Scheme. In *Proceedings of NOSSDAV*, Cambridge, UK., 1998.

[29] Unknown. Fast-start vs Streaming. http://www.apple.com/quicktime/.

[30] W. Willinger, M. S. Taqqu, R. Sherman, and D. V. Wilson. Self-similarity through high-variability: statistical analysis of Ethernet LAN traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[31] D. Wu. Streaming Video over the Internet: Approaches and Directions, 2001.

[32] N. Yeadon. *Quality of Service Filters for Multimedia Communications*. PhD thesis, Lancaster University, Lancaster, May 1996.

APPENDIX L

A Domain Specific Language for Component Configuration. Mark P Jones. Internal Draft for distribution to PCES TDs, May 8, 2002.

A Domain Specific Language for Component Configuration

Mark P Jones
Department of Computer Science & Engineering
OGI School of Science & Engineering
Oregon Health & Science University

Introduction

This note describes a *domain specific language* (DSL) for component configuration in the Boeing *Open Experimental Platform* (OEP). The purpose of the DSL is to make it easier for system integrators to construct and validate configurations from concise, modular, and reusable high-level descriptions. The design of the DSL reflects common patterns, terminology, and notations used in the specific domain, which in this case have to do with initializing software components, and establishing connections between them. By providing direct support for domain specific idioms, a DSL empowers its users to express their ideas quickly and concisely, to work more productively, to avoid certain kinds of coding error, and to tackle more complex problems than might otherwise be possible. The DSL that we describe here, for instance, has been used to produce a clear and modular description of the largest example in the current OEP build that is approximately 25 times smaller than the original description written in XML.

Component Configuration in Build 1.6.1 of the Boeing OEP

The Boeing OEP is a set of software applications that has been provided for use in the MoBIES and PCES programs as a platform for experimentation and testing of new software development tools and technologies. The OEP provides a library of configurable software components that are representative of the components used in aircraft control, navigation, and tracking systems together with configuration tools and a CORBA-based run-time infrastructure.

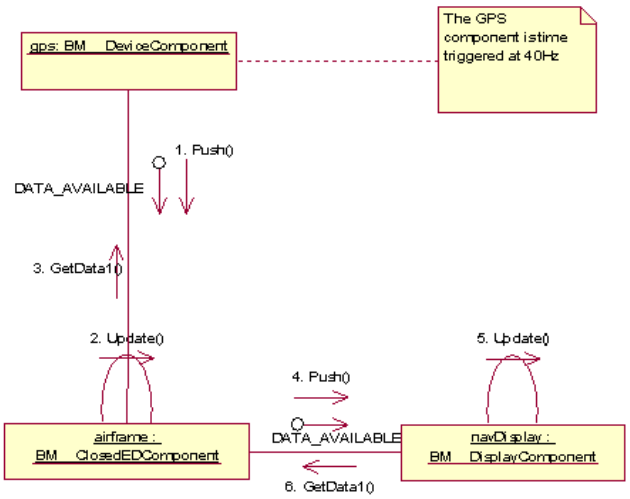
In Build 1.6.1 of the OEP, component configurations are described by means of hand-written XML data files, and a custom tool is used to turn these descriptions into executable C++ code.

The build includes several examples of these XML configuration files. These files encode a collection of product scenarios, each of which is described independently and at a high-level in a separate document using a mixture of prose and graphical notations. These examples are quite small and are intended primarily as illustrations to help researchers develop a better understanding of the domain. Even the largest example, referred to as Scenario 1.4, has only 50 components; the corresponding XML file is about 120K bytes long, comprising 3285 lines of XML data. By comparison, a more realistic example that is representative of systems being flown today (or planned for future development) could involve hundreds or even thousands of components. Scaling the techniques used in the OEP examples to a point where they can be used with these larger examples is clearly a significant challenge.

There are several reasons why the XML descriptions are quite large, even for simple configurations. First and foremost, the systems being described are fundamentally complex; they involve many components and rely on sophisticated an intricate patterns of communication. The verbose, textual representation of data XML also contributes significantly to the size of the configuration files. Manipulating these large files by hand using generic text or XML editors is difficult and error-prone. For example, in studying the XML data for Scenario 1.4, we uncovered several different kinds of errors—from simple typos to errors in the structuring of XML data, redundant code, and inconsistencies between the original written specification of the system and its representation in XML. None of these bugs, however, were easy to find and indeed, in our opinion, the number of errors found was surprisingly low given the size and complexity of the specification, and the relatively early stage of development.

One significant source of complexity arises from the need to map between the high-level view at which the prose/graphical descriptions of the components are specified in the OEP documentation, and the overall structure of the configuration files, which reflects more directly the mechanisms of component configuration in the context of the CORBA component model (CCM). In Figure 1, we see some indication of the semantic gap that must be bridged by the translation between these two views. The diagram on the left of the figure is a collaboration diagram describing the simplest example in the OEP, which is referred to as Scenario 1.1. In this

example, a `gps` object is triggered at 40Hz, and communicates in push-pull style with an `airframe` object that, in turn, drives a display unit, `navDisplay`.




```
<?xml version="1.0" encoding="UTF-8"?>
<CONFIGURATION>
  <PROCESSOR>
    <NAME> BM__PROCESSOR1 </NAME>
    <CONFIGURATION_PASS>
      <HOME> ... component type id ...
        <COMPONENT>
          ... individual component
                declarations ...
        </COMPONENT>
      </HOME>
      ...
    </CONFIGURATION_PASS>
    ...
  </PROCESSOR>
</CONFIGURATION>
```

Figure 1:Different views of Component Configuration for Scenario 1.1

The text on the right of the figure shows a fragment of the XML code for the same example. (This has been heavily trimmed; the original comprises 195 lines of XML.) Clearly, this version of the specification is organized in a very different manner, broken down at the top-level with one section for each processor in the system (in this case, there is only one processor, which is referred to as `BM__PROCESSOR1`). For each processor, the configuration script specifies multiple configuration passes (typically, two) that describe how to construct, initialize and connect the components that it will host. Passes are, in turn, described as sequences of "homes," each of which groups together all the components of a particular type. For example, all `BM__DeviceComponent` objects, such as `gps`, will be placed in one home, while all `BM__ClosedEDComponent` objects, such as `airframe`, will be described in another. The communication between the `gps` and `airframe` components in this example is reflected by declaring `gps` as an event supplier in the `BM__DeviceComponent` home, and by listing it again in an event consumer declaration for `airframe` in the home for `BM__ClosedEDComponent`. In general, these declarations should be placed in separate configuration passes to meet an OEP requirement that all event suppliers are defined before any of the matching event consumers. In the terminology of aspect-oriented programming, these examples illustrate the classic problems of *tangling*, in which a single high-level aspect at the design level becomes a cross-cutting concern in the XML description.

Tangling of a different kind occurs in the XML description of configurations that are constructed from a collection of smaller, independent subsystems. Scenario 1.2 in the OEP distribution is a simple example of this, as illustrated in Figure 2.

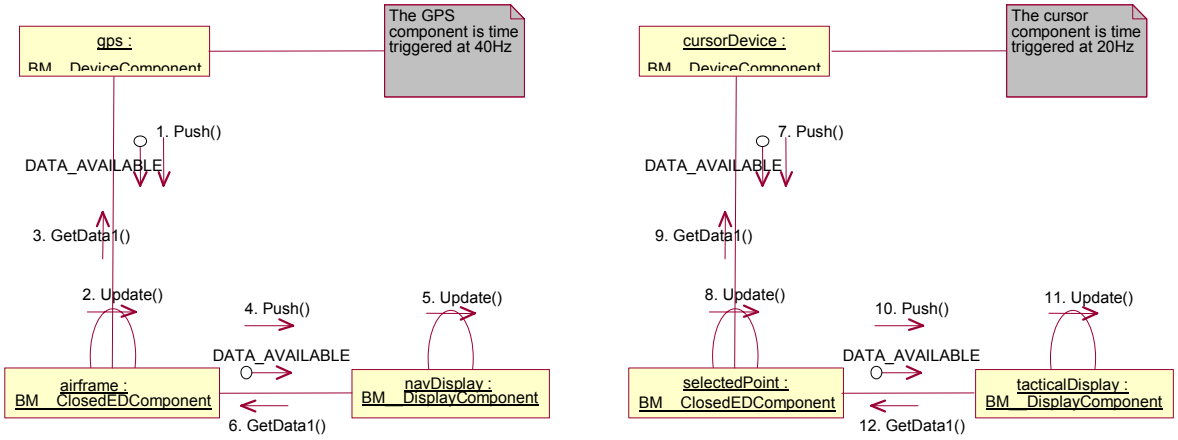


Figure 2: Collaboration Diagram for Scenario 1.2, comprising two similar but distinct subsystems

Here, the portion of the configuration on the left is just a copy of Scenario 1.1, while the portion on the right connects three new components in a very similar pattern. Indeed, the only difference between the two halves of this diagram, other that the choice of components, is that the left portion is triggered at twice the frequency of the left (40Hz versus 20Hz). Unfortunately, the XML description of Scenario 1.2 does not reflect these observations. Instead, it provides a monolithic view of the whole system, mixing together information about the left and right halves and making it harder to see the two pieces and independent subsystems, and harder still to see the structural similarities between the two halves. For example, information about the `gps` component is placed together with information about the `cursorDevice` component in the home for `BM__DeviceComponent` objects, even though these components serve completely independent roles in the full system. This weaving together of component descriptions is entirely appropriate for the purposes of runtime configuration, but also limits opportunities for modularity, reuse, and abstraction in the way that systems are described. Again, we see a semantic gap here, with the burden of translating between the design and runtime views of component configuration placed firmly in the hands of the authors of XML configurations.

There are also significant semantic bridges to be built (and crossed!) at lower levels. For example, in the OEP documentation, the product scenarios are described in terms of periodic signals that are used to trigger components at particular frequencies such as 1Hz, 10Hz, or 20Hz. In the corresponding XML files, however, these frequencies must be translated into time intervals such as 1000000, 100000, or 50000 (measured in micro-seconds). The author of each XML description of a configuration must perform this translation by hand, and, in this particular case, with special care to avoid errors; it is all too easy to enter the wrong number of trailing zeroes on these large numbers.

Another detail that makes it hard to find errors in the XML data is that the format for describing component configurations is not particularly well defined. The OEP distribution does include an XML *document type definition* (DTD) for configuration files, but it is intended only as a guide, and is followed only loosely in the examples. This gives the creators of the XML data more latitude and flexibility in describing a particular configuration, but also prevents the use of a *validating XML parser* to check that the XML meets simple structural constraints. Even with a DTD, however, there are still some important typing constraints that cannot be enforced. For example, event types are specified using an `EVENT_TYPE` element in the XML data whose content is a string that represents a single integer value, as in `<EVENT_TYPE>1000000</EVENT_TYPE>`. Entering some other string here would be an error, but nothing in the current process of authoring or processing the XML data will flag this as a problem. Of course there are more sophisticated methods for specifying XML files (XML Schema, for example) that can catch errors like this, but they still cannot address more global typing problems such as ensuring that an event consumer and the corresponding event sender use the same event type.

Towards a Domain Specific Language for The Boeing OEP

In the previous section, we highlighted several problems with the XML format that is used to describe component configurations in the OEP, at least from the perspective of somebody who must construct those descriptions by hand. In the past, similar observations about the difficulties of writing programs at a low-level using either machine code or assembly language prompted the development of new high-level languages, allowing programmers to focus more directly on the

problems that they needed to solve, and leaving details of the translation to a particular target machine to a compiler. Since then, the same strategy has been used in many times over, and the design and use of *domain specific languages (DSLs)* is widely recognized as a technique for improving programmer productivity and software reliability. A good DSL captures common patterns, terminology, idioms, and notations that are used in a specific domain, allowing domain experts to focus their attention on solving problems as quickly and effectively as possible, without being distracted by lower level concerns. Some of the most well-known and widely used DSLs include SQL (for describing database queries), yacc (for constructing parsers corresponding to context free grammars), HTML (for describing the content and layout of a web page), and LaTeX (for describing the structure and text in a typeset document). As these examples illustrate, DSLs are not designed as general purpose programming languages, but are instead designed to meet a specific need.

While the benefits of using a DSL may be appreciated, the cost of designing and implementing a new language of any kind can be quite high, and hard to justify in the context of projects whose deliverables are something other than new language processing tools. Economic pressures like these are especially strong in the case of DSLs for highly specialized domains where the potential market, or user base is very small.

Of course, there has been much work to develop technologies that will allow more rapid design, prototyping, experimentation, and application of new and effective DSLs. Building on a long history of work within the Pacific Software Research Center (PacSoft) at OGI, the Project Timber team brings extensive technical expertise in this area to the PCES program. In particular, the team is focused on the development of a new DSL called "*Timber*" that is designed to facilitate the construction and analysis of high-assurance, and portable real-time systems from high-level descriptions by encapsulating time-oriented and quality of service programming patterns. In fact, Timber is more than just a simple DSL; it is a foundation on which more specific DSLs can be built. For example, as a demonstration vehicle for the project's key technologies, we are using Timber to construct a DSL for programming applications on a mobile robot that involves both hard real-time control tasks and softer real-rate components such as video capture and streaming.

As we learned about the Boeing OEP, however, it became clear that we could also use Timber to build a new DSL for describing the configuration of components in the OEP. The resulting DSL will be described in the remaining sections of this paper. In fact, our OEP DSL does not make use of any of the new features that Timber provides to support explicit declaration of temporal behavior, reactive programming, or concurrency; while these are all important features of systems described by an OEP configurations, they are not properties of the configurations themselves. Nevertheless, the OEP DSL described here provides good evidence for the flexibility and expressiveness of Timber's core by virtue of its success in supporting a domain for which it was not originally designed. Of course, our current DSL reflects only a partial understanding of the domain; we have benefited already from the OEP documentation and from direct interaction with members of the Boeing team, but the DSL described here should only be considered a first prototype. We hope that by further interaction, we will be able to develop our DSL to the point where it provides a tool that the domain experts at Boeing can use to build larger configuration scenarios in a fraction of the time, and with less risk of errors than is possible using the current handwritten XML format.

Describing Scenario 1.1 using a DSL for Component Configuration

We will introduce the features of our OEP DSL informally, and in stages, using a series of examples. The syntax that we use reflects our current implementation, but many aspects could be changed quite easily, as needed, to provide a better fit with the notation used by domain experts. Our first example is a description of Scenario 1.1, previously illustrated in Figure 1:

```
scenario11 = do processor "BM__PROCESSOR1"

               -- This is where we define the components in this scenario
               airframe   <- new closedED(100, 110, "AIRFRAME")
               gps        <- new device(200, 220, "GPS")
               navDisplay <- new display(300, 330, "NAV_DISPLAY")

               -- This is where we describe the connections between them
               trigger 20      ==> gps
               event0 gps      ==> airframe    # fullChannel
               event1 airframe ==> navDisplay  # fullChannel
```

The first line here specifies a name for the configuration (`scenario11`) and begins its definition with the keyword `do`, which begins a sequence of commands. The sequence of commands splits naturally into three sections:

- The first section, comprising the single command `processor "BM__PROCESSOR1"`, sets the name of the processor for subsequent commands. This, of course, corresponds directly to uses of the `<PROCESSOR>` tag in the original XML description.

- The second section begins with a comment[1] and then three lines to declare the `gps`, `airframe`, and `navDisplay` components that play a role in this scenario. Each declaration specifies a component type (such as `device`, `closedED`, or `display`—each of which is defined elsewhere in a reusable library) together with group id and item id numbers and a human readable name string that can be used to identify the component at run-time. The id numbers used here were chosen to match the corresponding numbers in the XML description. We believe that it might be possible to generate appropriate id numbers automatically, in which case this aspect of our component description could be further simplified. (It is certainly possible to do this within our DSL; the question is whether this is appropriate to the domain and, if so, what scheme should be applied to generate suitable id numbers.) This example also illustrates the syntax, `val <- cmd`, for obtaining the return value, `val`, of a command, `cmd`. The command `new device(200, 220, "GPS")` could be used by itself to add a new component labeled `"GPS"` to a configuration, but there would be no way to refer to it again in the rest of the configuration without the handle, `gps`, that the call to `new` provides.

- The final section describes how the individual components are connected together. The first line, `trigger 20 ==> gps`, specifies that the `gps` component should be triggered at a frequency of 20 Hz. As the notation suggests, the `trigger 20` portion of this command is used to describe the timing signal, while the reference to gps on the right of the `==>`

[1] The two characters "`--`" begin a single line comment. Multi-line comments, beginning with "`{-`" and terminated by a corresponding "`-}`", are also supported.

operator specifies the receiver. The last two lines use a different mechanism to specify an event source: in a style that is (by design) reminiscent of variable declarations in C, C++ or Java, the notation `event0 gps` specifies that the `gps` component generates events of type `event0`, which is a standard event type and is again declared separately in a reusable library. As the remaining portion of this example describes, the `gps` events are sent to the `airframe`, which, in turn, passes events of type `event1` to the `navDisplay`. The `==>` operator used here captures a push-pull style of communication between components that is used throughout the OEP. As such, it specifies the source component as an event supplier and the receiver as an event consumer, but it also adds a receptacle that the consumer can use to query the supplier. This allows the supplier to send a relatively lightweight "DATA_AVAILABLE" event to the consumer, which can then query the consumer selectively for any further information that it requires. A final comment is needed to explain the `# fullChannel` annotation on these last two lines, which describes an aspect of Scenario 1.1 that is not captured explicitly in the diagram shown in Figure 1. To avoid cluttering commands with unnecessary details, our OEP DSL assumes default settings for some configuration parameters. In this case, the `fullChannel` annotation specifies that events should be delivered using the full event channel implementation that is included as part of the OEP infrastructure. Without this addition, the mechanism for event delivery would be left unspecified in the configuration, and would instead be set by the event channel at runtime.

As the length of this description suggests, our short DSL program provides a lot of information about the configuration of components in Scenario 1.1, but it is still quite readable. More importantly, by construction, it localizes the description of a single connection to a single point in the program; it avoids unnecessary repetition of details (for example, component ids are specified only once; subsequent references to a component use the handle returned by `new`); and it prevents errors such as an event type mismatch between suppliers and consumers, or the declaration of an event supplier with no consumers.

When an OEP DSL program is executed, it builds a database that describes the complete component configuration. Working within the Timber interpreter, we can use a command of the

form `writeXML "config.xml" scenario11` to run our program and use the resulting database to generate an XML configuration file, `config.xml`, which is suitable for input to the OEP tools. We can also view the XML directly on screen using the command `showMe scenario11`. Other uses of the configuration database could be accommodated in a similar way. Possible applications include static analysis tools and checkers that can verify critical properties or search a configuration for errors; optimization tools that detect and remove sources of inefficiency or unnecessary redundancy; and visualization tools that can generate graphical views of a particular configuration automatically, both for documentation and for visual confirmation of correctness.

Modularity, Reuse, and Abstraction in the OEP DSL

One simple but important detail that we mentioned only in passing during our presentation of Scenario 1.1 is the fact that we were able to give the program a name, `scenario11`. This is a simple form of *abstraction*—a mechanism that allows programmers to name and use fragments of code, or more general patterns within code—and an important first step towards increased software reuse and modularity. Here, for example is a description of Scenario 1.2 that builds directly on the definition of Scenario 1.1:

```
scenario12 = do scenario11      -- reuse the definition of Scenario 1.1

                selectedPoint   <- new closedED(400, 440, "SELECTED_POINT")
                cursorDevice    <- new device(500, 550, "CURSOR_DEVICE")
                tacticalDisplay <- new display(600, 660, "TACTICAL_DISPLAY")

                trigger 10           ==> cursorDevice
                event0 cursorDevice  ==> selectedPoint   # fullChannel
                event1 selectedPoint ==> tacticalDisplay # fullChannel
```

The first line here references the original description of Scenario 1.1; as a result, when we execute `scenario12`, we will automatically run the commands that are needed to build `scenario11`. The remaining lines add the new components for Scenario 1.2, and the connections between them using the same notations as before. The structure of this definition makes it much easier to see how Scenario 1.2 inherits from Scenario 1.1, while also maintaining a clear separation between the two distinct subsystems. Moreover, if some details of Scenario 1.1 need to be changed at a later date, then the definition of Scenario 1.2 will also inherit those modifications,

simplifying program maintenance, and avoiding the need to duplicate details across multiple configurations.

In other respects, the definition of `scenario12` is straightforward, and follows the same pattern as we saw in `scenario11`. Such similarities are unlikely to be accidental, and can often be found in situations where the same basic design pattern is used over and over again. By abstracting out the key structure as a reusable chunk of code, we can again produce more modular descriptions that are easier to maintain and reuse. In this particular situation, we can recognize a common pattern in the way that groups of three components are strung together in a pipeline, and driven by a time triggered event signal. In our OEP DSL, this pattern can be described as follows:

```
pipe3 freq dev filt disp = do trigger freq ==> dev
                              event0 dev   ==> filt # fullChannel
                              event1 filt  ==> disp # fullChannel
```

Here, `pipe3` gives a name to the whole pattern, while `freq`, `dev`, `filt`, and `disp` are the parameters[2] that can be expected to vary from one instantiation to the next. With this definition in place, we can rewrite the previous definitions of `scenario11` and `scenario12` to use our new `pipe3` abstraction:

```
scenario11a = do processor "BM__PROCESSOR1"
                 airframe    <- new closedED(100, 110, "AIRFRAME")
                 gps         <- new device(200, 220, "GPS")
                 nav_display <- new display(300, 330, "NAV_DISPLAY")
                 pipe3 20 gps airframe nav_display

scenario12a = do scenario11a
                 selectedPoint   <- new closedED(400, 440, "SELECTED_POINT")
                 cursorDevice    <- new device(500, 550, "CURSOR_DEVICE")
                 tacticalDisplay <- new display(600, 660, "TACTICAL_DISPLAY")
                 pipe3 10 cursorDevice selectedPoint tacticalDisplay
```

This is a small example, and the pattern that we have abstracted is very simple so the immediate benefits of the transformation used here are quite modest. In larger examples, however, this facility for abstracting and generalizing over code patterns like this becomes more important. Our use of abstraction reduces the number of lines of code, but more importantly, it collapses a sequence of commands into a single conceptual unit that is easier to recognize and understand.

[2] Timber uses an "applicative" syntax for function parameters, which means that parentheses are not needed when the argument to a function is a simple value like a constant or a variable name.

Other Features of the OEP DSL

The example programs given in previous sections have already explained the key mechanisms of the OEP DSL, but we will continue here and in the next section to introduce some of the remaining features by showing how they are used to encode some of the other scenarios from the OEP documentation, culminating, in the next section, in a compact rendering of Scenario 1.4.

We start, however, with Scenario 1.6, which is designed to illustrate how configurations can be designed to work in environments where some physical components—such as a sensor device on an aircraft—may or may not be present. From the perspective of component configuration, the basic structure here looks much the same as in Scenario 1.1, except that a different type of component is used for the `airframe`, and a new `ins` (inertial navigation system) component is added as an event supplier. (The `ins` is defined as a `cyclicDevice`, which is a special component provided by the OEP infrastructure that alternates between valid and invalid states at a frequency of 0.5Hz to simulate a component that is not always operational.)

```
scenario16 = do processor "OCP_P1"
                airframe <- new prioritizedEvent(100, 110, "AIRFRAME")
                disp     <- new display(200, 220, "DISPLAY")
                gps      <- new device(300, 330, "GPS")
                ins      <- new cyclicDevice(400, 440, "INS")

                trigger 20 ==> ins
                trigger 20 ==> gps
                [dataAvail ins, dataAvail gps] ==> airframe  # fullChannel
                dataAvail airframe             ==> disp      # fullChannel
```

The `dataAvail` event mentioned here is a new event type that OEP components use to send a `DATA_AVAILABLE` signal. New event types can be introduced by specifying the integer code that will used to represent the event at runtime (corresponding to the contents of an `<EVENT_TYPE>` element in an XML configuration file).

```
dataAvail = event 1000000
```

The `event` constructor used here will only allow an integer constant—such as `1000000`—to be used as an event identifier; any attempt to use a different type of value will trigger a diagnostic and prevent the user from running the program until the error has been corrected. Moreover, this is the only place in our programs where we will use the constant, `1000000`; in all other places, we

refer to the event by the name `dataAvail`, so there is little chance that we will mistakenly use different identifiers for the same event type in different parts of our program.

The other new feature in `scenario16` appears in the last but one line of the definition, which describes a connection from a list `[dataAvail ins, dataAvail gps]` of event suppliers to the `airframe` consumer. In fact our OEP DSL provides lists as a built-in type, together with a collection of operators that allow programmers to construct and manipulate list values. The `map` operator, for example, can be used to apply a function `f` to each element of a list: The expression `map f [e1, …, en]` is just another way of writing the list `[f e1, …, f en]`, and hence the list of event suppliers mentioned previously can also be written as `map dataAvail [ins, gps]`. More generally, we can define a new operator, `+=>`, as a variant of `==>`, that connects a list of suppliers, each producing a `dataAvail` event, to a consumer:

```
suppliers +=> consumer   =   map dataAvail suppliers ==> consumer
```

With this definition in place, we can rewrite the definition of `scenario16` as follows:

```
scenario16 = do processor "OCP_P1"
                airframe <- new prioritizedEvent(100, 110, "AIRFRAME")
                disp     <- new display(200, 220, "DISPLAY")
                gps      <- new device(300, 330, "GPS")
                ins      <- new cyclicDevice(400, 440, "INS")

                trigger 20 ==> ins
                trigger 20 ==> gps
                [ins, gps] +=> airframe      # fullChannel  -- this line has been changed
                dataAvail airframe ==> disp  # fullChannel
```

It is also useful to specify that the same event supplier (or list of suppliers) should be connected to multiple receivers. In the code for `scenario16`, for example, both the `ins` and `gps` components expect to receive a timing signal at 20Hz. We can capture this pattern with a general "fanout" operator, `==<`, which is another general-purpose variation on the `==>` operator. (The "`<`" character at the end of the name `==<` was chosen as a visual reminder of operator's fanout behavior.) It is actually quite easy to define this operator within our DSL although a detailed explanation is beyond the scope of this report; for now it suffices to notice that the definition is very concise:

```
suppliers ==< consumers   =   mapM_ (suppliers ==>) consumers
```

(This definition is, in fact, included as a standard part of our DSL implementation.) This fanout operator can now be used to provide a new version of the definition for Scenario 1.6:

```
scenario16 = do processor "OCP_P1"
                airframe <- new prioritizedEvent(100, 110, "AIRFRAME")
                disp     <- new display(200, 220, "DISPLAY")
                gps      <- new device(300, 330, "GPS")
                ins      <- new cyclicDevice(400, 440, "INS")

                trigger 20 ==< [ins, gps]                  -- this line has been changed
                [ins, gps] +=> airframe      # fullChannel
                dataAvail airframe ==> disp  # fullChannel
```

For a final variation on `scenario16`, we note that the last two commands in the definition specify a particular method of event delivery using the same `fullChannel` annotation. Instead of repeating the same annotation on each command in a given sequence, we can apply a single annotation to a whole sequence by prefixing a `do` expression with the annotation, as in the following, and final version of Scenario 1.6:

```
scenario16 = do processor "OCP_P1"
                airframe <- new prioritizedEvent(100, 110, "AIRFRAME")
                disp     <- new display(200, 220, "DISPLAY")
                gps      <- new device(300, 330, "GPS")
                ins      <- new cyclicDevice(400, 440, "INS")

                trigger 20 ==< [ins, gps]
                fullChannel (do [ins, gps] +=> airframe          -- changes here
                                dataAvail airframe ==> disp)     -- and here …
```

So far we have only seen one annotation, `fullChannel`, but there are several others that can be used in the same way, including `via ERM`—an alternative delivery mechanism to `fullChannel`—and `correlate`—used to request correlation of events. We will see uses of both in the next section.

To conclude this section, we present, without detailed comment, a rendering of Scenario 1.7 from the OEP documentation; all of the constructs used here have been introduced in the previous examples and should now seem quite familiar.

```
scenario17 = do processor "OCP_P1"
                airframe  <- new closedED(100, 110, "AIRFRAME")
                disp      <- new display(200, 220, "DISPLAY")
                gps       <- new device(300, 330, "GPS")
                flir      <- new cyclicDevice(400, 440, "FLIR")
                trackFile <- new extrapolate(500, 550, "TRACKFILE")
```

```
trigger 20 ==> gps
trigger 1  ==> flir
fullChannel (do dataAvail gps       ==> airframe
                [airframe, flir]    +=> trackFile
                dataAvail trackFile ==> disp)
```

We have now seen how our DSL can be used to provide high-level, readable descriptions for each of Scenarios 1.1, 1.2, 1.6, and 1.7. It is particularly important to note that the descriptions shown here are complete, resulting in XML configuration files that contain all the details of the original, handwritten versions. Despite this, however, the DSL versions of these scenarios are much more compact: a total of less than forty lines of code, compared with 1,198 lines of XML in the corresponding files from Build 1.6.1.

Describing Scenario 1.4 using the OEP DSL

To conclude, we present a description of Scenario 1.4 from the OEP documentation using the DSL for component configuration that we have described in this report. Scenario 1.4 is the largest example in the current build, involving 50 different components, and described by 3,285 lines of XML. In fact, Scenario 1.4 consists of two separate subsystems—one for navigation and one for tracking—and our top-level description captures this structure directly.

```
scenario14 = do processor "BM__PROCESSOR1"
                navigation
                tracking
```

The navigation subsystem is described by the code in Figure 3; in the OEP documentation, it was described by a collaboration diagram, which we reproduce here in Figure 4. The first line of the code for the navigation system uses the "`via`" operator in an annotation that specifies that events within this portion of the configuration should be delivered using the Event Registration Manager (ERM). This is an optimization that has been implemented in the Boeing OEP to avoid the overheads of full channel delivery when event suppliers and consumers are on the same process and in the same thread. Several other delivery modes can be specified using `via`; in fact the `fullChannel` annotation that we used previously is just an abbreviation for `via FULL_CHANNEL`.

```
navigation = via ERM
  (do wp1  <- new passive(7100, 7110, "WAYPOINT1")              -- Create 10 waypoints
      wp2  <- new passive(7200, 7210, "WAYPOINT2")
      wp3  <- new passive(7300, 7310, "WAYPOINT3")
      wp4  <- new passive(7400, 7410, "WAYPOINT4")
      wp5  <- new passive(7500, 7510, "WAYPOINT5")
      wp6  <- new passive(7600, 7610, "WAYPOINT6")
      wp7  <- new passive(7700, 7710, "WAYPOINT7")
      wp8  <- new passive(7800, 7810, "WAYPOINT8")
      wp9  <- new passive(7900, 7910, "WAYPOINT9")
      wp10 <- new passive(7101, 7111, "WAYPOINT10")

      earthModel <- new pushDatasrc(6100, 6110, "EARTH_MODEL")  -- Build an earth model,
      trigger 1  ==> earthModel                                 -- triggered once a
      earthModel <== [wp1, wp2, wp3, wp4, wp5,                  -- second and connected
                      wp6, wp7, wp8, wp9, wp10]                 -- to the waypoints

      leg1 <- new lazyActive(101, 111, "LEG1")                  -- Create 5 legs and
      leg2 <- new lazyActive(102, 112, "LEG2")                  -- hook them up to
      leg3 <- new lazyActive(103, 113, "LEG3")                  -- the waypoints
      leg4 <- new lazyActive(104, 114, "LEG4")
      leg5 <- new lazyActive(105, 115, "LEG5")
      [wp1,wp2,wp3] +=> leg1
      [wp3,wp4,wp5] +=> leg2
      [wp5,wp6,wp7] +=> leg3
      [wp7,wp8,wp9] +=> leg4
      [wp9,wp10]    +=> leg5

      route <- new openED(4700, 4710, "ROUTE")                  -- Feed the leg events
      [leg1,leg2,leg3,leg4,leg5] +=> route                      -- into a route object

      let src -=> dst = dataAvail src ==> dst                   -- Create remaining parts
          pipe cs = sequence_ (zipWith (-=>) cs (tail cs))      -- of navigation system

      flightPlan        <- new openED(4800, 4810, "FLIGHT_PLAN")
      pilotPrefs        <- new openED(4900, 4910, "PILOT_PREFS")
      waypointSteering  <- new modal(9200, 9210, "WAYPOINT_STEERING")
      flightplanDisplay <- new display(3400, 3410, "FLIGHTPLAN_DISPLAY")

      pipe [route, flightPlan, waypointSteering, flightplanDisplay]
      flightPlan <== pilotPrefs  -- N.B. pilotPrefs does not generate events!

      groundPoints <- new closedED(5500, 5510, "GROUND_POINTS")
      navSteering  <- new modal(9100, 9110, "NAV_STEERING")
      navDisplay2  <- new display(3500, 3510, "NAV_DISPLAY2")

      pipe [route, groundPoints, navSteering, navDisplay2]

      pilotControls <- new modeSource(8100, 8110, "PILOT_CONTROLS")
      pilotControls <== [waypointSteering, navSteering]
  )
```

Figure 3: DSL Code for the Navigation Subsystem of Scenario 1.4

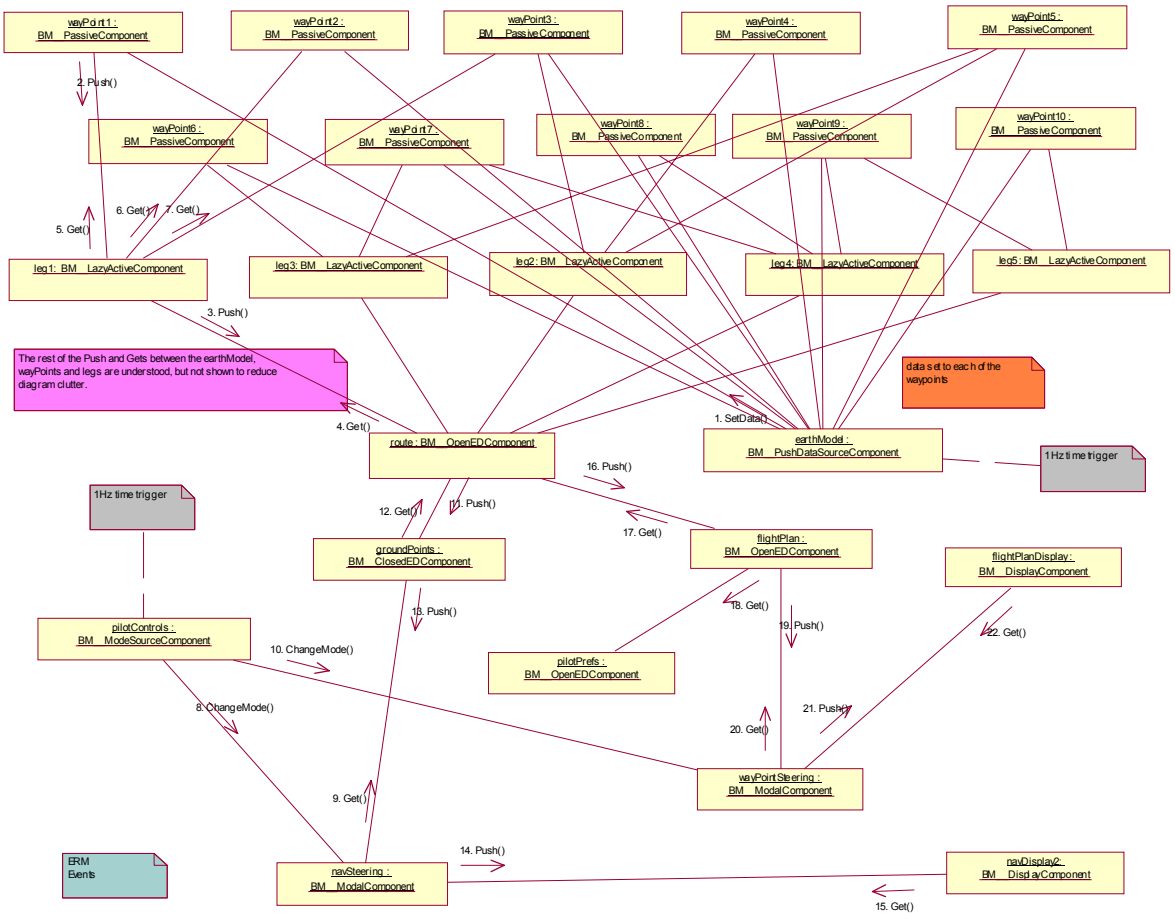


Figure 4: Collaboration Diagram for the Navigation Subsystem of Scenario 1.4

The other new feature in the navigation code is the <== operator, which we use to indicate a pull-style communication between components. In a connection of the form `event` c_1 ==> c_2, it is the left hand component c_1 that initiates communication by pushing an event to the right hand component c_2. In a connection of the form c_1 <== c_2, it is again the left hand component c_1 that initiates the communication, but this time by pulling data from the right hand component c_2. As such, there is no event type associated with this style of communication; the command c_1 <== c_2 simply defines a receptacle for c_2 (which is essentially a kind of interface pointer or callback for the component) on c_1. Our DSL allows either a single component or a list of components to be used on the right of the <== operator. The latter form is used in `navigation` to establish a connection between the `earthModel` and the ten `waypoint` objects, while the former adds a

receptacle for `pilotPrefs` to `flightPlan`. In fact, in studying the XML description of this scenario in Build 1.6.1, we discovered that `pilotPrefs` was, in addition declared as an event supplier, even though there were no consumers for its events. This does not change the functional behavior of the system at runtime, but it does incur an unnecessary overhead in generating and subsequently discarding the redundant events. The constructs in our DSL, by contrast, do not allow us to configure a component as an event supplier without an associated event consumer.

A careful look at the code for `navigation` reveals that approximately half of the lines in the definition are required simply to construct the 25 components in this subsystem; this was necessary to ensure that we used the same component and group id codes that are used in the XML description of Scenario 1.4 in Build 1.6.1. In future versions of our DSL, we will allow these id codes to be generated automatically, and provide a `multiNew` operator for constructing a list of components in a single command. This will allow us to reduce the first 26 lines in the definition of `navigation` to the following code:

```
wps <- multiNew passive[1..10]("WAYPOINT")

earthModel <- new pushDatasrc("EARTH_MODEL")
trigger 1  ==> earthModel
earthModel <== wps

legs <- multiNew lazyActive[1..5](“LEG”)
let wpSets = takeWhile (not.null) (map (take 3) (iterate (drop 2) wps))
mapM_ (zipWith (+=>) wpSets legs)

route <- new openED("ROUTE")
legs +=> route
```

More important than the reduction in code size, however, it is easier to modify this revised version of the code to accommodate a different numbers of `waypoint` and `leg` objects; we need only change the constants (or even replace them with parameters), and would not need to change the overall structure of the code by adding new declarations or commands.

The remaining portion—the `tracking` subsystem—for our description of Scenario 1.4 is shown in Figure 5, with the corresponding collaboration diagram from the OEP documentation reproduced in Figure 6. The only new DSL feature here is the `correlate` annotation, which is used to specify that the event consumer will only receive an event when all of the event suppliers have fired.

```
tracking = via FULL_CHANNEL
  (do gps    <- new device(200, 220, "GPS")              -- Create some devices,
      ins    <- new device(300, 330, "INS")              -- triggered at 20 Hz
      adc    <- new device(400, 440, "ADC")
      radar1 <- new device(500, 550, "RADAR1")
      radar2 <- new device(1000, 1100, "RADAR2")
      ts1    <- new device(600, 660, "TRACKSENSOR1")
      ts2    <- new device(700, 770, "TRACKSENSOR2")
      ts3    <- new device(800, 880, "TRACKSENSOR3")
      ts4    <- new device(900, 990, "TRACKSENSOR4")

      trigger 20 ==< [gps, ins, adc, radar1, radar2, ts1, ts2, ts3, ts4]

      airframe <- new lazyActive(100, 110, "AIRFRAME")    -- Create an airframe
      [gps, ins, adc, radar1] +=> airframe # correlate

      track1  <- new openED(4200, 4210, "TRACK1")         -- Create a collection
      track2  <- new openED(4300, 4310, "TRACK2")         -- of track objects
      track3  <- new openED(4400, 4410, "TRACK3")
      track4  <- new openED(4500, 4510, "TRACK4")
      track5  <- new closedED(5000, 5010, "TRACK5")
      track6  <- new openED(4600, 4610, "TRACK6")
      track7  <- new closedED(5100, 5110, "TRACK7")
      track8  <- new closedED(5200, 5210, "TRACK8")
      track9  <- new closedED(5300, 5310, "TRACK9")
      track10 <- new closedED(5400, 5410, "TRACK10")

      [ts1, ts2, ts3] +=> track1 # correlate             -- The track sensors trigger
      [ts1, ts2]      +=> track2 # correlate             -- the track objects in
      [ts1]           +=> track3                         -- curious ways
      [ts2]           +=> track4
      [ts3]           +=> track5
      [ts4]           +=> track6
      [ts1, ts2]      +=> track7 # correlate
      [ts3, ts4]      +=> track8 # correlate
      [ts4]           +=> track9
      [ts4]           +=> track10

      -- Tactical steering takes (collated) input from various sensors:
      tacticalsteering <- new openED(4000, 4010, "TACTICALSTEERING")
      [airframe, track1, track2, track3, track4, track5,
       track6, track7, track8, track9, track10, radar2] +=> tacticalsteering # correlate

      hud              <- new display(3100, 3110, "HUD")  -- add and connect displays
      tacticaldisplay1 <- new display(3200, 3210, "TACTICALDISPLAY1")
      tacticaldisplay2 <- new display(3300, 3310, "TACTICALDISPLAY2")
      navDisplay       <- new display(3000, 3010, "NAV_DISPLAY")

      dataAvail airframe ==> navDisplay
      dataAvail tacticalsteering ==< [hud, tacticaldisplay1, tacticaldisplay2]
  )
```

Figure 5: DSL Code for the Tracking Subsystem of Scenario 1.4

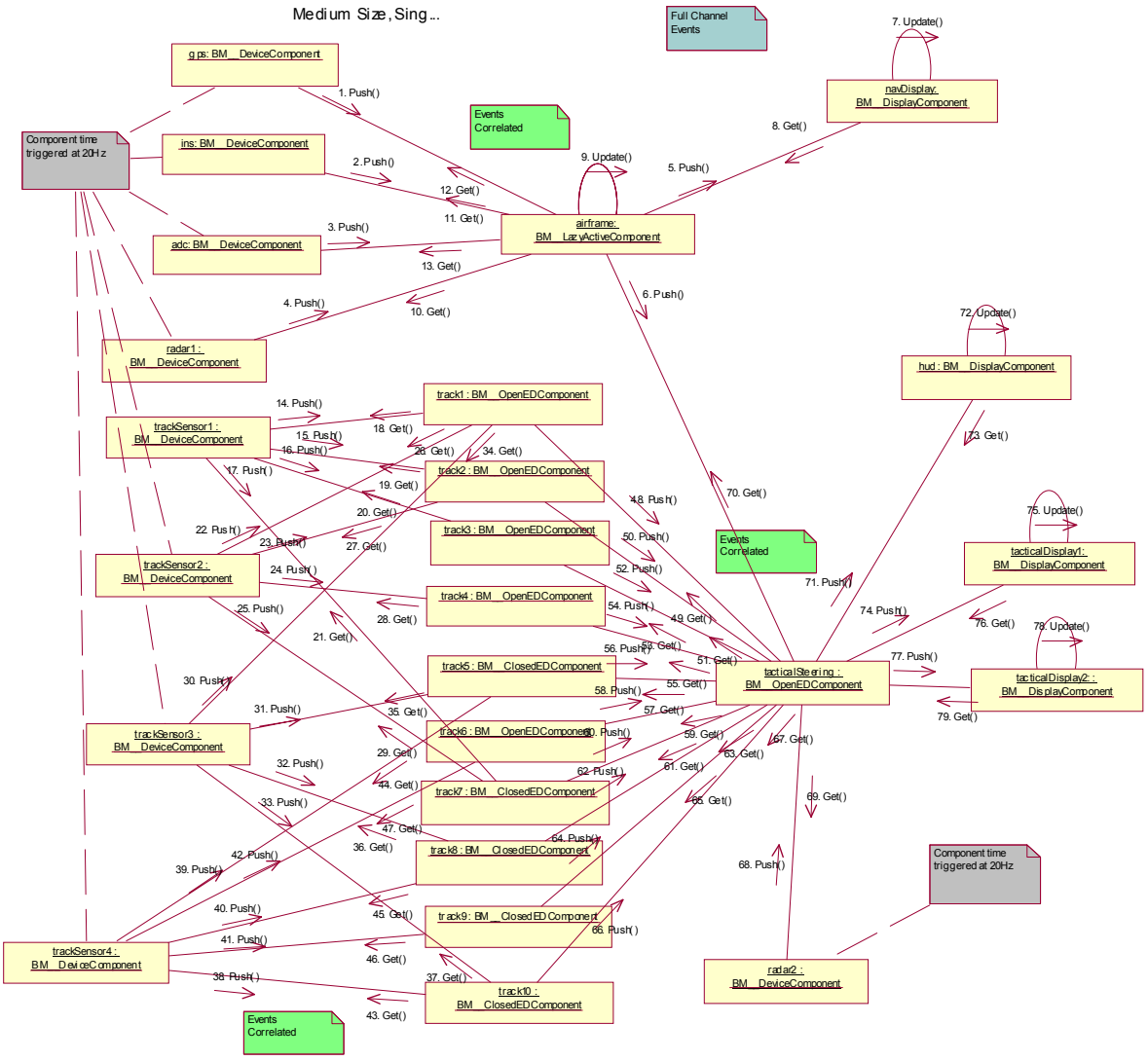


Figure 6: Collaboration Diagram for the Tracking Subsystem of Scenario 1.4

Conclusions

This report represents the beginning of an effort to explore the potential for using a domain specific language to simplify and support the construction of component configurations in the Boeing OEP, increasing the productivity of users while also reducing the potential for bugs. Our current prototype provides high-level constructs that allow configurations to be described in a

readable and compact notation, with significant opportunities for abstraction, modularity, and reuse. As a demonstration of feasibility, we have shown that our DSL can comfortably describe the largest example in the current OEP Build in less than 1/25th of the number of lines of code in the original XML version.

Of course, there is still a lot of work to be done, and we fully expect that we will need to make changes—from adjustments in syntax to more substantive modifications in key abstractions—as we learn more about the domain and the needs of users through our interactions with the OEP developers at Boeing. That said, the examples presented here show that our approach has much to offer, and we hope that it will soon be possible to exploit this work by using our DSL to describe, construct, and test some of the more complex scenarios that are planned for future OEP releases.

APPENDIX M

A Domain Specific Language for Component Configuration: Preliminary User Notes. Mark P. Jones. Documentation distributed with releases of the DSL software. Summer 2003.

A Domain Specific Language for Component Configuration: Preliminary User Notes

Mark P. Jones
Department of Computer Science & Engineering
OGI School of Science & Engineering at OHSU
Beaverton, Oregon 97006

1 Introduction

This note describes the current status, from a user perspective, of the domain specific language (DSL) that we have built for component configuration in the Boeing Open Experimental Platform (OEP). The purpose of the DSL is to make it easier for system integrators to construct and validate configurations from concise, modular, and reusable high-level descriptions. The design of the DSL reflects common patterns, terminology, and notations used in the specific domain, which in this case have to do with initializing software components, and establishing connections between them. By providing direct support for domain specific idioms, a DSL empowers its users to express their ideas quickly and concisely, to work more productively, to avoid certain kinds of coding error, and to tackle more complex problems than might otherwise be possible. The DSL that we describe here, for instance, has been used to produce a clear and modular description of the largest example in the current OEP build that is approximately 30 times smaller than the original description written in XML.

We assume that the reader is already familiar with the Boeing OEP and with the goals of our DSL in that specific context. Further information on these topics may be found elsewhere [1]. The OEP's native XML format for describing configurations is described in the Configuration Interface documentation [3].

2 An Overview of DSL Syntax and Notation

The DSL uses a textual notation that is designed to provide a close correspondence with the diagrams and the text used in the product scenario documentation [2]. We will introduce the notation using a small but complete example (Scenario 1.1, to be specific), whose text is as follows:

```
import OEP

example = do processor "BM__PROCESSOR1"
             airframe   <- new closedEDComp("AIRFRAME")
             gps        <- new deviceComp("GPS")
             navDisplay <- new displayComp("NAV_DISPLAY")

             trigger 20 gps

             gps      <=> airframe
             airframe <=> navDisplay
```

In the interests of being concrete, we will assume that this program has been stored in a plain text file called `example.hs` in the same directory as the `OEP.hs` file from the DSL distribution. Note that the interpreter is a bit fussy about layout, so it is important to get the indentation right here. In particular, each of the commands following the `do` keyword should begin in the same column. Commands may be broken over multiple lines as necessary, providing that all of the continuation lines are indented more than the first line of the command. On the other hand, the only reason for lining up the arrows, `<-`, was to make the code look"prettier." In practice, you don't need to align them like this unless you want to!

Every DSL program begins with the line `import OEP`, which initializes the system by loading the libraries that define the commands of the DSL. The rest of each DSL program is a sequence of definitions, each of which takes the form:

```
scenarioName = do command_1
                  command_2
                  ...
                  command_n
```

The main types of command that can be used are described in Section 2.1. Where needed, additional details and attributes are specified using simple annotations, each beginning with a single # character, The different forms of annotation are described in Section 2.2. The more esoteric commands having to do with concurrency and distribution are described in Section 2.3. Finally, Section 2.4 discusses a larger example that illustrates how these features are used together to build up complete descriptions of interesting configurations.

2.1 DSL Commands

This section provides a brief summary of each of the different kinds of command that can be used in a DSL description of a configuration.

2.1.1 Processor Name

A `processor "NAME"` command specifies the name of the processor that will host the components defined in subsequent commands (up to the next `processor` command). Every DSL program begins with a `processor` command. A configuration with multiple processors uses multiple `processor` commands, as in:

```
name = do processor "BM__PROCESSOR1"
          ...definition of components on processor 1...
          ...connections between components on processor 1...

          processor "BM__PROCESSOR2"
          ...definition of components on processor 2...
          ...connections between components on processor 2...

          ...connections between components
             on processor 1 and processor 2...
```

2.1.2 Component Construction

A command of the form `comp <- new componentType("NAME")` defines a new component of the specified `componentType` with label `"NAME"`. In sub-

DSL name	Name in Avionics OEP
closedEDComp	BM__CLOSED_ED_COMPONENT
deviceComp	BM__DEVICE_COMPONENT
displayComp	BM__DISPLAY_COMPONENT
lazyActiveComp	BM__LAZY_ACTIVE_COMPONENT
modalSourceComp	BM__MODE_SOURCE_COMPONENT
modalComp	BM__MODAL_COMPONENT
pushDatasrcComp	BM__PUSH_DATASRC_COMPONENT
passiveComp	BM__PASSIVE_COMPONENT
openEDComp	BM__OPEN_ED_COMPONENT
pushPullComp	BM__PUSH_PULL_COMPONENT
prioritizedEventComp	BM1__PRIORITIZED_EVENT_COMPONENT
cyclicDeviceComp	BM1__CYCLIC_DEVICE_COMPONENT
extrapolateComp	BM1__EXTRAPOLATE_COMPONENT
displaySurfaceComp	OM1__DISPLAY_SURFACE_COMPONENT
formatComp	OM1__FORMAT_COMPONENT
weaponDeliveryComp	OM1__WEAPON_DELIVERY_COMPONENT
dataGatheringComp	BM1__DATA_GATHERING_COMPONENT
waypointsComp	RM1__WAYPOINTS_COMPONENT
tracksComp	RM1__TRACKS_COMPONENT
routesComp	RM1__ROUTES_COMPONENT
routeComp	RM1__ROUTE_COMPONENT
pathComp	RM1__PATH_COMPONENT
userInputComp	BM__USER_INPUT_COMPONENT

Figure 1: Component types used in the OEP DSL

sequent commands the component should be referred to by the name `comp`. The list of component types supported in the current version of the OEP DSL is shown in Figure 1, together with the corresponding names that are used in the generated XML files, In the current implementation, these component types are predefined as part of the DSL library, which makes them easy to use, but also makes it harder to add support for new types of component. We expect to address this in a future version of the DSL.

Components can be defined in any order, and component definitions can be interleaved with other commands. The only restriction is that every compo-

nent should be defined before it is used. For example, the commands from the example at the beginning of Section 2 can be reordered as follows without changing the configuration that is described.

```
gps <- new deviceComp("GPS")
trigger 20 gps

airframe <- new closedEDComp("AIRFRAME")
gps <=> airframe

navDisplay <- new displayComp("NAV_DISPLAY")
airframe <=> navDisplay
```

Note that the DSL does not check for cases where two or more component definitions specify the same `NAME`, but this could easily be supported in a future version of the DSL.

2.1.3 Triggers and Timeouts

A command of the form `trigger freq comp` specifies that the `comp` component should be triggered at a frequency of `freq` Hz. (The `freq` value is typically 1, 5, 10, 20, or 40.) In the earlier example, for instance, the `gps` component is triggered at 20Hz. This corresponds to the `INTERVAL_TIMEOUT` tag that is used in the XML, except that it uses a frequency instead of a time interval. In fact there are several variations on this including:

- `trigger freq [comp_1, ..., comp_n]`

  Triggers each of the specified components at the given frequency. This is just a shorthand for:

  ```
  trigger freq comp_1
  ...
  trigger freq comp_n
  ```

- `timeout interval comp`

  `timeout interval [comp_1, ..., comp_n]`

These work just like the corresponding `trigger` commands, except that they use a time interval (in microseconds) instead of a frequency (in hertz).

These examples illustrate a syntax that is used throughout the DSL to denote a list of items: enclosing (square) brackets that delimit a comma separated sequence of values (in most cases, these are component names).

2.1.4 Connecting Components

There are three basic command forms that can be used to describe how components should be connected together:

- `comp ==> comp` creates a "push" connection from `l` to `r`. It indicates that `l` can generate data available events, and that these should be passed on to `r`.
- `comp <=> comp` creates a "push/pull" connection between `l` and `r`. It indicates that `l` can send a data available event to `r`, and that `r` will have a receptacle so that it can call back to get data from `l`.
- `comp <== comp` creates a "pull" connection between `l` and `r`. It indicates that `l` has a receptacle that it can use to get data from `r`.

Once you know the intuitions here, these notations should become quite easy to read (and remember). In each case:

- The left component is (typically) the one that initiates a communication, either by sending an event or by using a get state call.
- The arrow heads indicate the direction of data flow.

The push and push/pull commands all assume the common (hence default) case of a data available event type. It also possible to specify a different event type using the following variants:

```
event n   comp ==> comp
event n   comp <=> comp
```

In each case, `n` is an event type number. (Data available is event type `1000000`, for example.) We have added extra space here between the `n` and the first `comp`, but this is for clarity only, and is not required.

If the same event type is used multiple times, then it may be more convenient to define an abbreviation so that the number `n` does not need to be repeated. This can be done with a command of the form:

```
let myEvent = event 1000001
```

and then in subsequent commands, you can write commands like:

```
myEvent comp ==> comp
myEvent comp <=> comp
```

You can also define a new event at the top level of a DSL program but the `let` keyword should be omitted in this case. For example, the OEP includes the following definition for the data available event:

```
dataAvail = event 1000000
```

This means that you can use `dataAvail` as an event name in DSL code, if you wish, without having to define it yourself. The following two commands, for example, are equivalent:

```
gps           <=> airframe
dataAvail gps <=> airframe
```

There are also variants of `==>` and `<=>` for communications with multiple senders (using the general notation for lists mentioned previously):

```
[comp_1, ..., comp_m] ==> comp
[event n_1 comp_1, ..., event n_m comp_m] ==> comp
[ev_1 comp_1, ..., ev_m comp_m] ==>
```

The first of these is used when there are multiple suppliers of data available events, while the second and third forms are used when the suppliers provide different event types. There are similar command forms for `<=>`, of course.

Finally, we can use a `toEach` command to specify multiple connections with the same supplier or set of suppliers. For example:

```
toEach (comp ==>) [comp_1, ..., comp_n]
```

is a shorthand for the following sequence of commands:

```
comp ==> comp_1
...
comp ==> comp_n
```

The `toEach` construct can also be used with the `<=>` or `<==` connectors in place of `==>`, and with multiple suppliers and/or different types of event suppliers, by reusing the notation shown above. For example, the following command specifies a push/pull communication between `tacticalsteering` and each of the three components on the right hand side.

```
toEach (tacticalsteering<=>)
   [hud, tacticaldisplay1, tacticaldisplay2]
```

2.2 DSL Annotations

Special properties of components are described by annotations. In the simplest case, an annotation is written after a command, with a single `#` character between the command and the annotation, as in:

```
command # annotation
```

It is also possible to provide multiple annotations, as in:

```
command # annotation_1
        # annotation_2
```

Note that the DSL syntax for annotations is quite specific, and you should not use an annotation to attach an arbitrary textual comment to part of a DSL program. In the DSL, a single line textual comment begins with the two characters `--` and extends to the end of that line.

```
command                -- comment
  # annotation         -- text comment
  # another annotation -- more comment
```

Several different forms of annotation are supported:

- A `correl` annotation is used to indicate that events should be correlated. For example, the command:

  ```
  [gps, ins, adc, radar1] <=> airframe # correl
  ```

  describes a push/pull communication from the four components on the left whose data available events will be correlated before the `airframe` component on the right receives them. If `correl` is not specified then a default of uncorrelated is assumed.

- The `ERM_EVENT` and `FULL_EVENT` annotations correspond to the different methods of sending events (via ERM, or through the full event channel, respectively), as in:

  ```
  [gps, ins, adc, radar1] <=> airframe # FULL_EVENT
  ```

- An `internalLock n` or `externalLock n` annotation should be added to the definition of a component to specify the type of locking that is required, as in the following example:

  ```
  routes <- new pushPullComp("ROUTES") # externalLock 2
  ```

  The default, if no explicit locking annotation is given, is equivalent to `internalLock 2`.

- Persistence attributes of a component can be set using an annotation of the form:

  ```
  persistent{regionID=r, saveMethodType=s,
             classified=c, trackDirtyness=t,
             savePolicy=sp, doubleBuffered=d}
  ```

  Any or all of the six fields can be omitted, in which case the defaults shown in the following annotation will be used:

  ```
  persistent{regionID=1, saveMethodType=1,
             classified=0, trackDirtyness=0,
             savePolicy=FilterTime 0, doubleBuffered=0}
  ```

If no persistent annotation is specified, then the component will not have any persistence properties. To specify a component with the default persistence properties shown above, you can just use the `persistent` annotation by itself without any modifiers, as in:

```
steering <- new closedEDComp("STEERING") # persistent
```

To specify persistence properties but override one or more of the defaults shown above, simply list the desired bindings (in any order) between braces, as in the following example:

```
comp <- new compType("NAME") # persistent{classified=1}
```

- Internal attributes can be specified using annotations, as in the following example:

```
hudFormat <- new formatComp("HUD_FORMAT")
   # internal [Item "FORMAT_ID" "1",
               Item "START_TIME" "1",
               Item "INTERVAL" "3"]
```

These `internal` annotations correspond to a relatively new feature in the OEP, and we expect the notation to evolve as both the OEP and the DSL mature. In particular, the need for such annotations may one day be avoided by folding appropriate, type-specific alternatives into the syntax for constructing `new` components.

- An `updateRate f` annotation specifies an update rate for a communication or a master-proxy relationship. The `f` parameter is the required frequency in hertz (typical values are 1Hz, 5Hz, 10Hz, 20Hz, and 40Hz, but any integer frequency can be specified).

- There are also `static`, `dynamic`, and `distWrite` annotations, but these are only useful with the `proxy` command, and so will be described in Section 2.3.

Sometimes it is useful to apply an annotation to a group of commands. The DSL provides a `with` construct to support this in examples like the following:

```
with annotation
 (do command_1
     ...
     command_n)
```

This is almost equivalent to:

```
command_1   # annotation
...
command_n   # annotation
```

(The only exception is that, in the first version, the names of any components that are defined in any one of the commands are local to that list of commands, and cannot be referenced outside of the block.)

The following fragment from Scenario 1.2 illustrates how `with` is used in practice, here describing a pipelined sequence of connections between four components, with all events passed via the full event channel implementation:

```
with FULL_EVENT
 $ do gps <=> airframe
      cursorDevice <=> selectedPoint
      airframe <=> navDisplay
      selectedPoint <=> tacticalDisplay
```

This example also illustrates a minor variant in syntax that you may see in some of the DSL code generated by the `xml2dsl` tool: instead of enclosing the whole `do command_1 ... command_n` block in parentheses, you can just insert a single `$` sign is in front of the `do` keyword. DSL users are free to use either parentheses or `$`, as they prefer!

2.3 Commands for Concurrency and Distribution

This section describes advanced DSL commands that were not discussed in Section 2.1. These are needed to describe some of the more esoteric features of certain configurations having to do with concurrency and distribution.

- A `synchProxyGroup` command is used to specify the components in a synchronous proxy group, together with a unique identifier for the group itself. The following example shows an example of such a command (taken from the DSL code for Scenario 1.5):

```
synchProxyGroup airframe "AIRFRAME_SmartPointer"
  [(airframeSynchproxy40hz,0), (airframeSynchproxy20hz,1),
   (airframeSynchproxy10hz,2), (airframeSynchproxy5hz,3),
   (airframeSynchproxy1hz,4)]
```

  The name `"AIRFRAME_SmartPointer"` specifies the identifier that will be used for this group. A `synchProxyGroup` command should only be used after all of the individual components involved have been defined (i.e., in this case, after the definitions of the main `airframe` component as well as each of the proxies `airframeSynchProxy40hz`, ...). Note that this command also specifies an integer code giving the update rate for each proxy. These codes use the values specified in the documentation for the OEP configuration interface [3, Section 2.6.10] instead of raw frequency values.

- A `sharedLock comp [comp_1, ... comp_n]` command is used to indicate that each of the listed components `comp_1`, ..., `comp_n` should share the lock of the named master component `comp`. For example, the following command is used to specify a shared lock in the DSL code for Scenario 2.1:

```
sharedLock routes [route0, route1, path]
```

- Last but not least, a `proxy comp compProxy` command is used to specify a master-proxy relationship between a component, `comp`, and a corresponding proxy, `compProxy`. This is used in multi-processor scenarios where the master and proxy are located on different processors. It is possible to have multiple proxies for a single master in which case the second argument of the `proxy` command should be a list, as in the following example from Scenario 4.1:

```
proxy airframe [airframeProxy, airframeProxy3]
```

Here, `airframe` is a master component on processor 1, `airframeProxy` is a proxy on processor 2, and `airframeProxy3` is another proxy, this time on processor 3.

Additional details about a master-proxy relationship can be specified using annotations on the `proxy` command. For example, a `dynamic` annotation should be used to specify where dynamic replication is required. (Alternatively, a `static` annotation can be used to specify static replication, but this is the default so it does not normally need to be spelled out explicitly.) In a similar way, the `distWrite` annotation should be used to specify when distributed writes are required. (Once again, the alternative, `noDistWrite`, is the default and does not usually need to be specified explicitly.) Finally, the `updateRate f` annotation, previously described in Section 2.2, can also be used on a `proxy` command to specify the update frequency for proxies. For example, the following command is used in the DSL code for Scenario 3.2 to specify a master-proxy relationship with distributed writes and an update rate of 1Hz:

```
proxy waypoint waypointProxy # distWrite
                             # updateRate 1
```

We encourage the reader to study the examples of DSL code in the distribution, particularly for the 3.x and 4.x scenarios, to see how these features are used in practice.

2.4 A Larger Example

One of the important features of the DSL is the support that it provides for breaking large, complex configurations into smaller, independent components. This enhances modularity and reuse, makes larger configurations easier to understand, and facilitates the construction of large configurations by a team of developers. This is essential in practice because it is unrealistic to expect that larger scenarios will always be produced by a lone developer.

With that in mind, this section presents the complete DSL code for Scenario 1.4 in the OEP product scenarios [2], whose overall structure is depicted in the diagram in Figure 2. It is clear from the figure that this particular

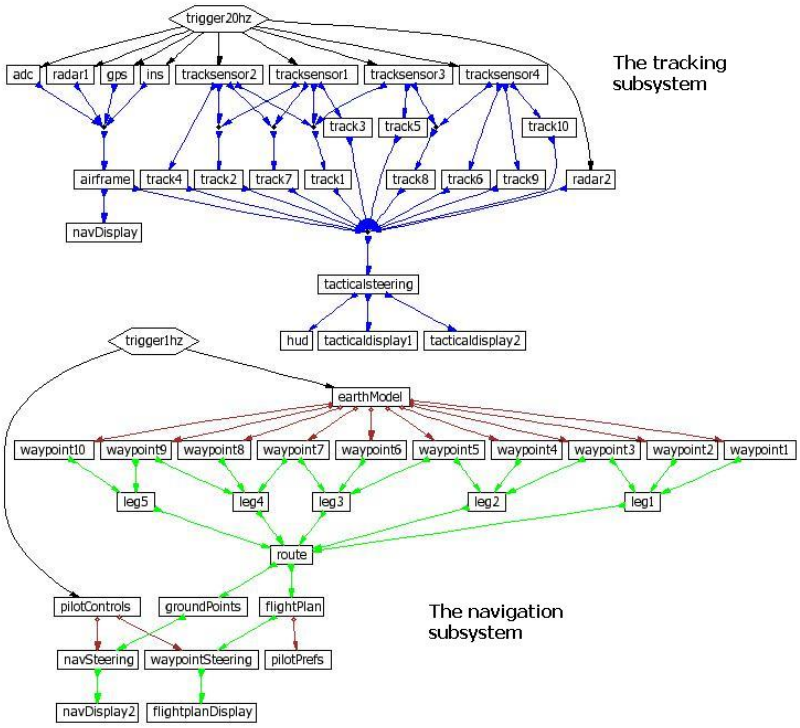


Figure 2: Graphical View of Scenario 1.4

configuration splits neatly into two separate pieces, one for tracking and one for navigation. This is an important property of the system that is not captured in the native XML format that is used to describe the configuration. Instead, the XML code merges the two subsystems into a monolithic chunk of data that arranges components according to type rather than functionality.

By contrast, the DSL allows Scenario 1.4 to be expressed naturally as a combination of the two subsystems. More specifically, we can create a DSL description for this scenario with a program that begins as follows:

```
import OEP

scenario14 = do processor "BM__PROCESSOR1"
                tracking   # FULL_EVENT
                navigation # ERM_EVENT
```

The names `tracking` and `navigation` used here are placeholders for sections of DSL code that could be written by different developers[1]. As it happens, all of the components in the `tracking` system use ERM for event delivery, while all of the components in the `navigation` system use the full event channel. This is another property of the configuration that is hard to see from the XML code, but is captured elegantly here by attaching an appropriate annotation to each of the subsystems.

Now our task is to provide DSL code for each of the subsystems. Starting with the `tracking` system at the top of Figure 2, we can see a small, intricately wired network of track sensor and track components, which we can describe as a separate unit using the following code:

```
makeTracks
 = do tracksensor1 <- new deviceComp("TRACKSENSOR1")
      tracksensor2 <- new deviceComp("TRACKSENSOR2")
      tracksensor3 <- new deviceComp("TRACKSENSOR3")
      tracksensor4 <- new deviceComp("TRACKSENSOR4")
      trigger 20 [tracksensor1, tracksensor2,
                  tracksensor3, tracksensor4]

      track1 <- new openEDComp("TRACK1")
      track2 <- new openEDComp("TRACK2")
      track3 <- new openEDComp("TRACK3")
      track4 <- new openEDComp("TRACK4")
      track6 <- new openEDComp("TRACK6")
      track5 <- new closedEDComp("TRACK5")
      track7 <- new closedEDComp("TRACK7")
      track8 <- new closedEDComp("TRACK8")
      track9 <- new closedEDComp("TRACK9")
      track10 <- new closedEDComp("TRACK10")

      [tracksensor1, tracksensor2, tracksensor3] <=> track1
        # correl
      toEach ([tracksensor1, tracksensor2]<=>) [track2, track7]
        # correl
```

[1]The definitions for these two subsystems would, in practice, be imported from separate source files, but we do not illustrate that here.

```
        tracksensor1 <=> track3
        tracksensor2 <=> track4
        toEach (tracksensor4<=>) [track6, track9, track10]
        tracksensor3 <=> track5
        [tracksensor3, tracksensor4] <=> track8 # correl
        return [track1,track2,track3,track4,track5,
                track6,track7,track8,track9,track10]
```

Although it is fairly long, this code is fairly easy to understand. First we create 4 track sensor objects and arrange for each of them to be triggered at 20Hz. Then we create 10 track objects and wire them to the sensors, using event correlation where needed, in the configuration shown in the diagram. The final result from this section of code is a list of the ten track objects, which is returned by the last line of code.

Now we can describe the construction and connections between the remaining components to complete the description of the `tracking` system.

```
tracking
 = do gps <- new deviceComp("GPS")
      ins <- new deviceComp("INS")
      adc <- new deviceComp("ADC")
      radar1 <- new deviceComp("RADAR1")
      radar2 <- new deviceComp("RADAR2")

      trigger 20 [gps, ins, adc, radar1, radar2]

      airframe <- new lazyActiveComp("AIRFRAME")
      [gps, ins, adc, radar1] <=> airframe # correl

      navDisplay <- new displayComp("NAV_DISPLAY")
      airframe <=> navDisplay

      hud <- new displayComp("HUD")
      tacticaldisplay1 <- new displayComp("TACTICALDISPLAY1")
      tacticaldisplay2 <- new displayComp("TACTICALDISPLAY2")
      tacticalsteering <- new openEDComp("TACTICALSTEERING")
      toEach (tacticalsteering<=>)
       [hud, tacticaldisplay1, tacticaldisplay2]
```

```
tracks <- makeTracks
([airframe, radar2]++tracks) <=> tacticalsteering
  # correl
```

This code is also fairly straightforward, with the components being introduced and connected in the order that they appear as we move down the figure. The only unusual feature here are in the last two commands. First, `tracks <- makeTracks` executes the `makeTracks` code described previously, using the variable `tracks` to record the list of track objects that it returns. Finally, the last line uses those tracks, together with events from the `airframe` and `radar2` components, to provide the trigger for `tacticalSteering`. (The symbol `++` used here denotes list concatenation.)

That completes our description of `tracking` ... but we still have work to do. In particular, we must now provide a description for the `navigation` subsystem. Again motivated by the layout of the components in Figure 2, we will describe this in two pieces, the first of which describes the routing subsystem from the `route` component upwards[2]:

```
routeSubsystem
 = do earthModel <- new pushDatasrcComp("EARTH_MODEL")
      trigger 1 earthModel

      waypoint1 <- new passiveComp("WAYPOINT1")
      waypoint2 <- new passiveComp("WAYPOINT2")
      waypoint3 <- new passiveComp("WAYPOINT3")
      waypoint4 <- new passiveComp("WAYPOINT4")
      waypoint5 <- new passiveComp("WAYPOINT5")
      waypoint6 <- new passiveComp("WAYPOINT6")
      waypoint7 <- new passiveComp("WAYPOINT7")
      waypoint8 <- new passiveComp("WAYPOINT8")
      waypoint9 <- new passiveComp("WAYPOINT9")
      waypoint10 <- new passiveComp("WAYPOINT10")
```

[2]Diagrams are a useful guide, but they do not necessarily contain all of the information that is needed to complete the definition of a configuration. From time to time, additional details must be obtained from the original product scenario descriptions [2] or by consulting a domain expert.

```
        [waypoint1, waypoint2, waypoint3,
         waypoint4, waypoint5, waypoint6,
         waypoint7, waypoint8, waypoint9,
         waypoint10] <== earthModel

        leg1 <- new lazyActiveComp("LEG1")
        leg2 <- new lazyActiveComp("LEG2")
        leg3 <- new lazyActiveComp("LEG3")
        leg4 <- new lazyActiveComp("LEG4")
        leg5 <- new lazyActiveComp("LEG5")
        [waypoint1, waypoint2, waypoint3] <=> leg1
        [waypoint3, waypoint4, waypoint5] <=> leg2
        [waypoint5, waypoint6, waypoint7] <=> leg3
        [waypoint7, waypoint8, waypoint9] <=> leg4
        [waypoint9, waypoint10]           <=> leg5

        route <- new openEDComp("ROUTE")
        [leg1, leg2, leg3, leg4, leg5] <=> route
        return route
```

The second subsystem describes the portion of the diagram below the `route` component. However, we do not want to repeat the construction of the `route` component, and so we arrange for it to be provided here as a parameter[3]. Other than that, the rest of the code should now be very easy to follow:

```
steeringSubsystem route
 = do flightPlan   <- new openEDComp("FLIGHT_PLAN")
      groundPoints <- new closedEDComp("GROUND_POINTS")
      toEach (route<=>) [flightPlan, groundPoints]

      waypointSteering  <- new modalComp("WAYPOINT_STEERING")
      flightplanDisplay <- new displayComp("FLIGHTPLAN_DISPLAY")
      flightPlan        <=> waypointSteering
      waypointSteering  <=> flightplanDisplay
```

[3]All definitions can be parameterized in the manner shown here by adding the names of the parameters on the left-hand side of the `=` symbol.

```
navSteering  <- new modalComp("NAV_STEERING")
navDisplay2  <- new displayComp("NAV_DISPLAY2")
groundPoints <=> navSteering
navSteering  <=> navDisplay2

pilotPrefs    <- new openEDComp("PILOT_PREFS")
pilotControls <- new modalSourceComp("PILOT_CONTROLS")
trigger 1 pilotControls

[navSteering, waypointSteering] <== pilotControls
pilotPrefs <== flightPlan
```

This completes our description of Scenario 1.4 using the DSL. In total, our code spans 121 lines, including some blank likes that have been included only to make the code easier to read. In fact the code would have been shorter still if we had not opted to break it into small subsystems. However, the overhead for doing that is modest in comparison to the potential benefits that it offers for increased modularity and reuse. For comparison, the XML description of the same configuration in the OEP build is 2,675 lines of code. By running the DSL code above, we can generate an alternative, but equivalent XML file with a tighter layout. Event then, however, the resulting XML spans over 1,081 lines, so the DSL code is still about an order of magnitude smaller. Generally speaking, our experience suggests that the ratio between lines of DSL code and lines of XML improves even further as the size of the scenario increases. In the case of Scenario 4.1, the biggest example in the current distribution, the DSL code is approximately 30 times smaller than the original XML.

2.5 Key Points

The specific details of what we did in Section 2.4 are not particularly important: our purpose is only to show the potential for decomposing the description of a large system into smaller, more easily digestible pieces. The choice of decomposition is in the hands of the developer, and can be made to suit the needs of a team and to reflect domain specific insights. Note also that the `tracking` and `makeTracks` code could be written by different developers. The only thing they need to agree on is the interface between the components (in this case, the fact that `makeTracks` will return a list of

track objects). Once the interface has been fixed, the two developers are free to work independently. For example, if the design team decides to change the number of track sensors, the number of tracks, or the wiring between them, only the code for `makeTracks` needs to be modified. Moreover, if the developers anticipate that the set of components described by `makeTracks` is likely to be useful in other configurations, then they can easily move its code into a library that can be shared with other project teams.

To further emphasize some of these points, we will now show how the pieces of code used to describe Scenario 1.4 can be reused to build variations on this configuration:

```
example1               -- only the tracking subsystem
 = do processor "BM__PROCESSOR1"
      with FULL_EVENT tracking

example2               -- only the navigation subsystem
 = do processor "BM__PROCESSOR1"
      with ERM_EVENT navigation

example3               -- using the event channel for both
 = do processor "BM__PROCESSOR1"            -- subsystems
      with ERM_EVENT (do navigation
                         tracking)

example4               -- a distributed version placing the
 = do processor "BM__PROCESSOR1" -- two subsystems on
      with ERM_EVENT navigation  -- distinct processors ...

      processor "BM__PROCESSOR2"
      with FULL_EVENT tracking
```

Note that each of these examples defines a complete configuration by assembling pieces of the original configuration in slightly different ways.

3 Loading and Executing DSL Programs

So now you have a complete DSL program stored in a file like the `example.hs` file mentioned above. What can you do with it? First of all, you'll need to make sure that it has access to the OEP DSL implementation in `OEP.hs`, and the easiest way to do that is to put it in the same directory as `OEP.hs`. (There are other/better ways to do this by making an appropriate path setting, but the details for doing this are likely to change rapidly as the tools mature so placing things in the same directory is an acceptable workaround for the time being.)

Now you can load the DSL example by double clicking on the `example.hs` file. Assuming there are no errors in the DSL code, you will soon be presented with a prompt, probably something like the following:

```
Main>
```

At this point, there are just a few commands that you might try:

- `writeXML "filename.xml" scenario`: The `scenario` argument should be replaced by the name of the sequence of commands that describe your configuration (such as `example`, `example1`, etc., if you are using the DSL code described in Section 2.4). This this dumps the XML for the specified `scenario` into the file called `filename.xml`. Take care, because this command will overwrite any existing file with the same name, without pausing for confirmation.
- `showMe scenario`: This will print the XML version of the specified `scenario` on the console. This is often quite a lot of text, and hence may only be useful for small examples; in most cases, the `writeXML` command mentioned previously will be more suitable.
- `writeDot "filename.dot" [] scenario`: This will generate a description of the communication dependency graph in a format that can be fed to the DOT tool for visualization. (The empty list `[]` used here can be replaced with a list of hints to the layout algorithms used in DOT.)

Further guidance on the use of the DSL is provided in the `index.html` file that is included in the current distribution.

References

[1] Mark P. Jones. A Domain Specific Language for Component Configuration. Technical Report 02-016, OGI School of Science & Engineering, Oregon Health & Science University, November 2000.

[2] Wendy Roll. Product Scenarios Description Document for the Weapon System Open Experimental Platform, Version 2.4. The Boeing Company, May 2003.

[3] Wendy Roll. Weapon Systems OEP Configuration Interface, Version 2.4. The Boeing Company, May 2003.